(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 November 2005 (03.11.2005)

PCT

(10) International Publication Number
WO 2005/103924 A1

(51) International Patent Classification[7]: G06F 15/16

(21) International Application Number: PCT/AU2005/000578

(22) International Filing Date: 22 April 2005 (22.04.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
2004902146 22 April 2004 (22.04.2004) AU

(71) Applicant *(for all designated States except US)*: **WARATEK PTY LIMITED** [AU/AU]; Suite 18, 12 Tyron Road, Lindfield, NSW 2070 (AU).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: **HOLT, John, Matthew** [AU/AU]; Suite 18, 12 Tyron Road, Lindfield, NSW 2070 (AU).

(74) Agent: **FRASER OLD & SOHN**; Patent Attorneys, Level 6, 118 Alfred Steet, Milsons Point, NSW 2061 (AU).

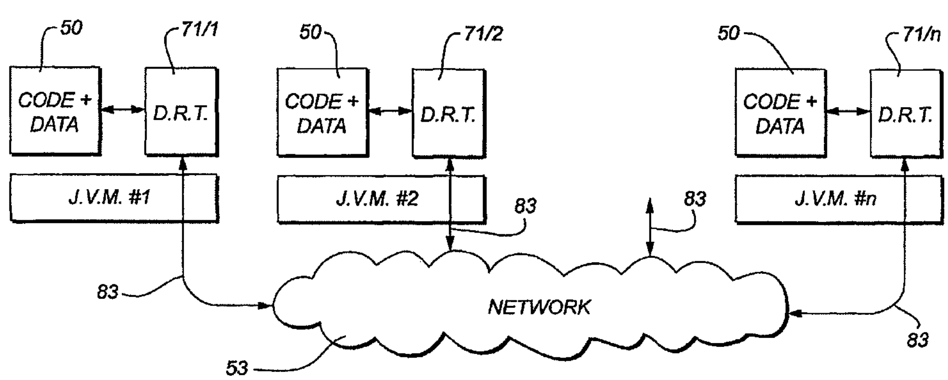
(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) **Title:** MODIFIED COMPUTER ARCHITECTURE WITH INITIALIZATION OF OBJECTS



(57) **Abstract:** The present invention discloses a modified computer architecture which (50, 71, 72) enables an applications program (50) to be run simultaneously on a plurality of computers (M1, ... Mn). Shared memory at each computer is updated with amendments and/or overwrites so that all memory read requests are satisfied locally. During initial program loading (75), or similar, instructions which result in memory being re-written or manipulated are identified (92). Additional instructions are inserted (103) to cause the equivalent memory locations at all computers to be updated. In particular, the initialisation of JAVA language classes and objects is disclosed (162, 163) so all memory locations for all computers are initialized in the same manner.

WO 2005/103924 A1

MODIFIED COMPUTER ARCHITECTURE WITH INITIALIZATION OF OBJECTS

Field of the Invention

The present invention relates to computers and, in particular, to a modified machine architecture which enables the operation of an application program simultaneously on a plurality of computers interconnected via a communications network.

Background Art

Ever since the advent of computers, and computing, software for computers has been written to be operated upon a single machine. As indicated in Fig. 1, that single prior art machine 1 is made up from a central processing unit, or CPU, 2 which is connected to a memory 3 via a bus 4. Also connected to the bus 4 are various other functional units of the single machine 1 such as a screen 5, keyboard 6 and mouse 7.

A fundamental limit to the performance of the machine 1 is that the data to be manipulated by the CPU 2, and the results of those manipulations, must be moved by the bus 4. The bus 4 suffers from a number of problems including so called bus "queues" formed by units wishing to gain an access to the bus, contention problems, and the like. These problems can, to some extent, be alleviated by various stratagems including cache memory, however, such stratagems invariably increase the administrative overhead of the machine 1.

Naturally, over the years various attempts have been made to increase machine performance. One approach is to use symmetric multiple processors. This prior art approach has been used in so called "super" computers and is schematically indicated in Fig. 2. Here a plurality of CPU's 12 are connected to global memory 13. Again, a bottleneck arises in the communications between the CPU's 12 and the memory 13. This process has been termed "Single System Image". There is only one application and one whole copy of the memory for the application which is distributed over the global memory. The single application can read from and write to, (ie share) any memory location completely transparently.

Where there are a number of such machines interconnected via a network, this is achieved by taking the single application written for a single machine and partitioning the required memory resources into parts. These parts are then distributed across a number of computers to form the global memory 13 accessible by all CPU's 12. This procedure relies on masking, or hiding, the memory partition from the single running application program. The performance degrades when one CPU on one machine must access (via a network) a memory location physically located in a different machine.

Although super computers have been technically successful in achieving high computational rates, they are not commercially successful in that their inherent complexity makes them extremely expensive not only to manufacture but to administer. In particular, the single system image concept has never been able to scale over "commodity" (or mass produced) computers and networks. In particular, the Single System Image concept has only found practical application on very fast (and hence very expensive) computers interconnected by very fast (and similarly expensive) networks.

A further possibility of increased computer power through the use of a plural number of machines arises from the prior art concept of distributed computing which is schematically illustrated in Fig. 3. In this known arrangement, a single application program (Ap) is partitioned by its author (or another programmer who has become familiar with the application program) into various discrete tasks so as to run upon, say, three machines in which case n in Fig. 3 is the integer 3. The intention here is that each of the machines M1...M3 runs a different third of the entire application and the intention is that the loads applied to the various machines be approximately equal. The machines communicate via a network 14 which can be provided in various forms such as a communications link, the internet, intranets, local area networks, and the like. Typically the speed of operation of such networks 14 is an order of magnitude slower than the speed of operation of the bus 4 in each of the individual machines M1, M2, etc.

Distributed computing suffers from a number of disadvantages. Firstly, it is a difficult job to partition the application and this must be done manually. Secondly,

communicating data, partial results, results and the like over the network 14 is an administrative overhead. Thirdly, the need for partitioning makes it extremely difficult to scale upwardly by utilising more machines since the application having been partitioned into, say three, does not run well upon four machines. Fourthly, in the event that one of the machines should become disabled, the overall performance of the entire system is substantially degraded.

A further prior art arrangement is known as network computing via "clusters" as is schematically illustrated in Fig. 4. In this approach, the entire application is loaded onto each of the machines M1, M2 ....Mn. Each machine communicates with a common database but does not communicate directly with the other machines. Although each machine runs the same application, each machine is doing a different "job" and uses only its own memory. This is somewhat analogous to a number of windows each of which sell train tickets to the public. This approach does operate, is scalable and mainly suffers from the disadvantage that it is difficult to administer the network.

In computer languages such as JAVA and MICROSOFT.NET there are two major types of constructs with which programmers deal. In the JAVA language these are known as objects and classes. Every time an object is created there is an initialization routine run known as "<init>". Similarly, every time a class is loaded there is an initialization routine known as "<clinit>". Other languages use different terms but utilize a similar concept.

The present invention discloses a computing environment in which an application program operates simultaneously on a plurality of computers. In such an environment it is necessary to ensure that the abovementioned initialization routines operate in a consistent fashion across all the machines. It is this goal of consistent initialization that is the genesis of the present invention.

In accordance with a first aspect of the present invention there is disclosed a multiple computer system having at least one application program running simultaneously on a plurality of computers interconnected by a communications network, wherein a like plurality of substantially identical objects are created, each in

the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

In accordance with a second aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and operating at least one application program simultaneously wherein each said computer in operating said at least one application program creates objects only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

In accordance with a third aspect of the present invention there is disclosed a method of running at least one application program on a plurality of computers simultaneously, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) creating the initial contents of each of said identically named objects substantially the same.

In accordance with a fourth aspect of the present invention there is disclosed a method of operating at least one application program simultaneously on a plurality of computers all interconnected via a communications link and each having at least a minimum predetermined local memory capacity, said method comprising the steps of:

(i) initially providing each local memory in substantially identical condition,

(ii) satisfying all requests to create objects generated by said application program in said local memory, and

(iii) communicating via said communications link all said objects created at each said computer and which reside locally to all the remainder of said plurality of computers whereby the objects of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

In accordance with a fifth aspect of the present invention there is disclosed a method of compiling or modifying an application program to run simultaneously on a plurality of computers interconnected via a communications link, said method comprising the steps of:

(i) detecting instructions which create objects utilizing one of said computers,

(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

In accordance with a sixth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program are simultaneously being processed each on a corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

In accordance with a seventh aspect of the present invention there is disclosed a method of ensuring consistent initialization of an application program to be run simultaneously on a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an initialization routine, and

(ii) modifying said initialization routine to ensure consistent operation of all said computers.

In accordance with a eighth aspect of the present invention there is disclosed a computer program product comprising a set of program instructions stored in a storage medium and operable to permit a plurality of computers to carry out the abovementioned methods.

Brief Description of the Drawings

Embodiments of the present invention will now be described with reference to the drawings in which:

Fig. 1 is a schematic view of the internal architecture of a conventional

computer,

Fig. 2 is a schematic illustration showing the internal architecture of known symmetric multiple processors,

Fig. 3 is a schematic representation of prior art distributed computing,

Fig. 4 is a schematic representation of a prior art network computing using clusters,

Fig. 5 is a schematic block diagram of a plurality of machines operating the same application program in accordance with a first embodiment of the present invention,

Fig. 6 is a schematic illustration of a prior art computer arranged to operate JAVA code and thereby constitute a JAVA virtual machine,

Fig. 7 is a drawing similar to Fig. 6 but illustrating the initial loading of code in accordance with the preferred embodiment,

Fig. 8 is a drawing similar to Fig. 5 but illustrating the interconnection of a plurality of computers each operating JAVA code in the manner illustrated in Fig. 7,

Fig. 9 is a flow chart of the procedure followed during loading of the same application on each machine in the network,

Fig. 10 is a flow chart showing a modified procedure similar to that of Fig. 9,

Fig. 11 is a schematic representation of multiple thread processing carried out on the machines of Fig. 8 utilizing a first embodiment of memory updating,

Fig. 12 is a schematic representation similar to Fig. 11 but illustrating an alternative embodiment,

Fig. 13 illustrates multi-thread memory updating for the computers of Fig. 8,

Fig. 14 is a schematic illustration of a prior art computer arranged to operate in JAVA code and thereby constitute a JAVA virtual machine,

Fig. 15 is a schematic representation of n machines running the application program and serviced by an additional server machine X,

Fig. 16 is a flow chart of illustrating the modification of initialization routines,

Fig. 17 is a flow chart illustrating the continuation or abortion of initialization routines,

Fig. 18 is a flow chart illustrating the enquiry sent to the server machine X,

Fig. 19 is a flow chart of the response of the server machine X to the request of Fig. 18,

Fig. 20 is a flowchart illustrating a modified initialization routine for the <clinit> instruction,

Fig. 21 is a flowchart illustrating a modified initialization routine for the <init> instruction,

Fig. 22 is a schematic representation of two laptop computers interconnected to simultaneously run a plurality of applications, with both applications running on a single computer,

Fig. 23 is a view similar to Fig. 22 but showing the Fig. 22 apparatus with one application operating on each computer, and

Fig. 24 is a view similar to Figs. 22 and 23 but showing the Fig. 22 apparatus with both applications operating simultaneously on both computers.

The specification includes Annexures A and B which provide actual program fragments which implement various aspects of the described embodiments. Annexure A relates to fields and Annexure B relates to initialization.

Detailed Description

In connection with Fig. 5, in accordance with a preferred embodiment of the present invention a single application program 50 can be operated simultaneously on a number of machines M1, M2...Mn communicating via network 53. As it will become apparent hereafter, each of the machines M1, M2...Mn operates with the same application program 50 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same application code and data 50. Similarly, each of the machines M1, M2...Mn operates with the same (or substantially the same) modifier 51 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same (or substantially the same) modifier 51 with the modifier of machine M2 being designated 51/2. In addition, during the loading of, or preceding the execution of, the application 50 on each machine M1, M2...Mn, each application 50 has been modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimising changes are permitted within each modifier 51/1 ...51/n).

As a consequence of the above described arrangement, if each of the machines M1, M2...Mn has, say, a shared memory capability of 10MB, then the total shared

memory available to each application 50 is not, as one might expect, 10n MB but rather only 10MB. However, how this results in improved operation will become apparent hereafter. Naturally, each machine M1, M2...Mn has an unshared memory capability. The unshared memory capability of the machines M1, M2...Mn are normally approximately equal but need not be.

It is known from the prior art to operate a machine (produced by one of various manufacturers and having an operating system operating in one of various different languages) in a particular language of the application, by creating a virtual machine as schematically illustrated in Fig. 6. The prior art arrangement of Fig. 6 takes the form of the application 50 written in the Java language and executing within a Java Virtual Machine 61. Thus, where the intended language of the application is the language JAVA, a JAVA virtual machine is created which is able to operate code in JAVA irrespective of the machine manufacturer and internal details of the machine. For further details see "The JAVA Virtual Machine Specification" 2nd Edition by T. Lindholm & F. Yellin of Sun Microsystems Inc. of the USA.

This well known prior art arrangement of Fig. 6 is modified in accordance with the preferred embodiment of the present invention by the provision of an additional facility which is conveniently termed "distributed run time" or DRT 71 as seen in Fig. 7. In Fig. 7, the application 50 is loaded onto the Java Virtual Machine 72 via the distributed runtime system 71 through the loading procedure indicated by arrow 75. A distributed run time system is available from the Open Software Foundation under the name of Distributed Computing Environment (DCE). In particular, the distributed runtime 71 comes into operation during the loading procedure indicated by arrow 75 of the JAVA application 50 so as to initially create the JAVA virtual machine 72. The sequence of operations during loading will be described hereafter in relation to Fig. 9.

Fig. 8 shows in modified form the arrangement of Fig. 5 utilising JAVA virtual machines, each as illustrated in Fig. 7. It will be apparent that again the same application 50 is loaded onto each machine M1, M2...Mn. However, the communications between each machine M1, M2...Mn, and indicated by arrows 83, although physically routed through the machine hardware, are controlled by the

individual DRT's 71/1...71/n within each machine. Thus, in practice this may be conceptionalised as the DRT's 71/1...71/n communicating with each other via the network 73 rather than the machines M1, M2...Mn themselves.

Turning now to Figs. 7 and 9, during the loading procedure 75, the program 50 being loaded to create each JAVA virtual machine 72 is modified. This modification commences at 90 in Fig. 9 and involves the initial step 91 of detecting all memory locations (termed fields in JAVA - but equivalent terms are used in other languages) in the application 50 being loaded. Such memory locations need to be identified for subsequent processing at steps 92 and 93. The DRT 71 during the loading procedure 75 creates a list of all the memory locations thus identified, the JAVA fields being listed by object and class. Both volatile and synchronous fields are listed.

The next phase (designated 92 in Fig. 9) of the modification procedure is to search through the executable application code in order to locate every processing activity that manipulates or changes field values corresponding to the list generated at step 91 and thus writes to fields so the value at the corresponding memory location is changed. When such an operation (typically putstatic or putfield in the JAVA language) is detected which changes the field value, then an "updating propagation routine" is inserted by step 93 at this place in the program to ensure that all other machines are notified that the value of the field has changed. Thereafter, the loading procedure continues in a normal way as indicated by step 94 in Fig. 9.

An alternative form of initial modification during loading is illustrated in Fig. 10. Here the start and listing steps 90 and 91 and the searching step 92 are the same as in Fig. 9. However, rather than insert the "updating propagation routine" as in step 93 in which the processing thread carries out the updating, instead an "alert routine" is inserted at step 103. The "alert routine" instructs a thread or threads not used in processing and allocated to the DRT, to carry out the necessary propagation. This step 103 is a quicker alternative which results in lower overhead.

Once this initial modification during the loading procedure has taken place, then either one of the multiple thread processing operations illustrated in Figs. 11 and 12 takes place. As seen in Fig. 11, multiple thread processing 110 on the machines consisting of threads 111/1...111/4 is occurring and the processing of the second

thread 111/2 (in this example) results in that thread 111/2 becoming aware at step 113 of a change of field value. At this stage the normal processing of that thread 111/2 is halted at step 114, and the same thread 111/2 notifies all other machines M2...Mn via the network 53 of the identity of the changed field and the changed value which occurred at step 113. At the end of that communication procedure, the thread 111/2 then resumes the processing at step 115 until the next instance where there is a change of field value.

In the alternative arrangement illustrated in Fig. 12, once a thread 121/2 has become aware of a change of field value at step 113, it instructs DRT processing 120 (as indicated by step 125 and arrow 127) that another thread(s) 121/1 allocated to the DRT processing 120 is to propagate in accordance with step 128 via the network 53 to all other machines M2...Mn the identity of the changed field and the changed value detected at step 113. This is an operation which can be carried out quickly and thus the processing of the initial thread 111/2 is only interrupted momentarily as indicated in step 125 before the thread 111/2 resumes processing in step 115. The other thread 121/1 which has been notified of the change (as indicated by arrow 127) then communicates that change as indicated in step 128 via the network 53 to each of the other machines M2...Mn.

This second arrangement of Fig. 12 makes better utilisation of the processing power of the various threads 111/1...111/3 and 121/1 (which are not, in general, subject to equal demands) and gives better scaling with increasing size of "n", (n being an integer greater than or equal to 2 which represents the total number of machines which are connected to the network 53 and which run the application program 50 simultaneously). Irrespective of which arrangement is used, the changed field and identities and values detected at step 113 are propagated to all the other machines M2...Mn on the network.

This is illustrated in Fig. 13 where the DRT 71/1 and its thread 121/1 of Fig. 12 (represented by step 128 in Fig. 13) sends via the network 53 the identity and changed value of the listed memory location generated at step 113 of Fig. 12 by processing in machine M1, to each of the other machines M2...Mn.

Each of the other machines M2...Mn carries out the action indicated by steps 135 and 136 in Fig. 13 for machine Mn by receiving the identity and value pair from the network 53 and writing the new value into the local corresponding memory location.

In the prior art arrangement in Fig. 3 utilising distributed software, memory accesses from one machine's software to memory physically located on another machine are permitted by the network interconnecting the machines. However, such memory accesses can result in delays in processing of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine. This in large part accounts for the diminished performance of the multiple interconnected machines.

However, in the present arrangement as described above in connection with Fig. 8, it will be appreciated that all reading of data is satisfied locally because the current value of all fields is stored on the machine carrying out the processing which generates the demand to read memory. Such local processing can be satisfied within $10^2 - 10^3$ cycles of the central processing unit. Thus, in practice, there is substantially no waiting for memory accesses which involves reads.

However, most application software reads memory frequently but writes to memory relatively infrequently. As a consequence, the rate at which memory is being written or re-written is relatively slow compared to the rate at which memory is being read. Because of this slow demand for writing or re-writing of memory, the fields can be continually updated at a relatively low speed via the inexpensive commodity network 53, yet this low speed is sufficient to meet the application program's demand for writing to memory. The result is that the performance of the Fig. 8 arrangement is vastly superior to that of Fig. 3.

In a further modification in relation to the above, the identities and values of changed fields can be grouped into batches so as to further reduce the demands on the communication speed of the network 53 interconnecting the various machines.

It will also be apparent to those skilled in the art that in a table created by each DRT 71 when initially recording the fields, for each field there is a name or identity which is common throughout the network and which the network recognises.

However, in the individual machines the memory location corresponding to a given named field will vary over time since each machine will progressively store changed field values at different locations according to its own internal processes. Thus the table in each of the DRTs will have, in general, different memory locations but each global "field name" will have the same "field value" stored in the different memory locations.

It will also be apparent to those skilled in the art that the abovementioned modification of the application program during loading can be accomplished in up to five ways by:

(i) re-compilation at loading,

(ii) by a pre-compilation procedure prior to loading,

(iii) compilation prior to loading,

(iv) a "just-in-time" compilation, or

(v) re-compilation after loading (but, or for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, eg from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

In the first embodiment, a particular machine, say machine M2, loads the application code on itself, modifies it, and then loads each of the other machines M1, M3 ... Mn (either sequentially or simultaneously) with the modified code. In this arrangement, which may be termed "master/slave", each of machines M1, M3, ... Mn loads what it is given by machine M2.

In a still further embodiment, each machine receives the application code, but modifies it and loads the modified code on that machine. This enables the modification carried out by each machine to be slightly different being optimized based upon its architecture and operating system, yet still coherent with all other similar modifications.

In a further arrangement, a particular machine, say M1, loads the unmodified code and all other machines M2, M3 ... Mn do a modification to delete the original application code and load the modified version.

In all instances, the supply can be branched (ie M2 supplies each of M1, M3, M4, etc directly) or cascaded or sequential (ie M2 applies M1 which then supplies M3 which then supplies M4, and so on).

In a still further arrangement, the machines M1 to Mn, can send all load requests to an additional machine (not illustrated) which is not running the application program, which performs the modification via any of the aforementioned methods, and returns the modified routine to each of the machines M1 to Mn which then load the modified routine locally. In this arrangement, machines M1 to Mn forward all load requests to this additional machine which returns a modified routine to each machine. The modifications performed by this additional machine can include any of the modifications covered under the scope of the present invention.

Persons skilled in the computing arts will be aware of at least four techniques used in creating modifications in computer code. The first is to make the modification in the original (source) language. The second is to convert the original code (in say JAVA) into an intermediate representation (or intermediate language). Once this conversion takes place the modification is made and then the conversion is reversed. This gives the desired result of modified JAVA code.

The third possibility is to convert to machine code (either directly or via the abovementioned intermediate language). Then the machine code is modified before being loaded and executed. The fourth possibility is to convert the original code to an intermediate representation, which is then modified and subsequently converted into machine code.

The present invention encompasses all four modification routes and also a combination of two, three or even all four, of such routes.

Turning now to Fig. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

In the JAVA language, the initialization routine <clinit> happens only once when a given class file 50A is loaded. However, the initialization routine <init> happens often, for example every time a new object 50X, 50Y and 50Z is created. In addition, classes are loaded prior to objects so that in the application program illustrated in Fig. 14, having a single class 50A and three objects 50X-50Z, the first class 50A is loaded first, then the first object 50X is loaded, then second object 50Y is loaded and finally third object 50Z is loaded. Where, as in Fig. 14, there is only a single computer or machine 72, then no conflict or inconsistency arises in the running of the initialization routines intended to operate during the loading procedure.

However, in the arrangement illustrated in Fig. 8, (and also in Figs. 22-24), a plurality of individual computers or machines M1, M2 ..... Mn are provided each of which are interconnected via a communications network 53 and each of which is provided with a modifier 51 and loaded with a common application program 50. Essentially the modifier 51 is to replicate an identical memory structure and contents on each of the individual machines M1, M2...Mn. It follows therefore that in such a computing environment it is necessary to ensure that each of the individual machines M1, M2...Mn is initialized in a consistent fashion. The modifying function of the modifier 51 of Fig. 5 is provided by the DRT 71 in Fig. 8.

In order to ensure consistent initialization, the application program 50 is scrutinized in order to detect program steps which define an initialization routine. This scrutiny can take place either prior to loading, or during the loading procedure 75, or even after the loading procedure 75 (but before execution of the relevant

corresponding application code). It may be likened to a compilation procedure with the understanding that the term compilation normally involves a change in code or language, eg from source to object code or one language to another. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include embrace modifications within the same code or language.

As a consequence, in the abovementioned scrutiny <clinit> routines are initially looked for and when found a modifying code (typically several instructions) is inserted so as to give rise to a modified <clinit> routine. This modified routine is to load the class 50A on one of the machines, for example JVM#1, and tell all the other machines M2...Mn that such a class 50A exists and its present state. There are several different modes whereby this modification and loading can be carried out.

Thus, in one mode, the DRT 71 on the loading machine, in this example JVM#1, asks the DRT's 71/2...71/n of all the other machines if the first class 50A has already been initialized. If the answer to this question is yes, then the normal initialization procedure is turned off or disabled. If the answer is no, then the normal initialization procedure is operated and the consequential changes brought about during that procedure are transferred to all other machines as indicated by arrows 83 in Fig. 8.

A similar procedure happens on each occasion that an object, say 50X, 50Y or 50Z is to be loaded. Where the DRT 71/1 does not discern, as a result of interrogation, that the particular object, say object 50Y, in question has already been loaded onto the other machines M2...Mn, then the DRT 71/1 runs the object initialization routine, and loads on each of the other machines M2...Mn an equivalent object (which may conveniently be termed a peer object) together with a copy of the initial values. However, if the DRT 71/1 determines that the object 50Y in question already exists on the other machines, then the normal initialization function is disabled and a local copy is created with a copy of the current values. Again there are various ways of bringing about the desired result.

As seen in Fig. 15 a modification to the general arrangement of Fig. 8 is provided in that machines M1, M2...Mn are as before and run the same application program (or programmes) 50 on all machines M1, M2...Mn simultaneously. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply a housekeeping function, and especially the initialisation of structures, assets and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in Fig. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably operated as dual machines in a cluster. The additional machine X+1 is optional as indicated by the broken lines in Fig. 15.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2...Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function.

Fig. 16 shows a preferred general procedure to be followed. After a loading step 161 has been commenced, the instructions to be executed are considered in sequence and all initialization routines are detected as indicated in step 162. In the JAVA language these are the <init> and <clinit> routines (or methods in JAVA terminology). Other languages use different terms.

Where an initialization routine is detected in step 162, it is modified in step 163, typically by inserting further instructions into the routine. Alternatively, the modifying instructions could be inserted prior to the routine. Once the modification step 163 has been completed the loading procedure continues, as indicated in step 164.

Fig. 17 illustrates a particular form of modification. After commencing the routine in step 171, the structures, assets or resources (in JAVA termed classes or objects) to be initialised are, in step 172, allocated a name or tag which can be used globally by all machines. This is most conveniently done via a table maintained by

server machine X of Fig 15. This table also includes the status of the class or object to be initialised.

As indicated in Fig. 17, if steps 173 and 174 determine that the global name is not already initialised elsewhere (ie on a machine other than the machine carrying out the loading) then this means that the object or class can be initialised in the normal fashion by carrying out step 176 since it is the first such object or class to be created.

However, if steps 173 and 174 determine that the global name is already utilised elsewhere, this means that another machine has already initialised this class or object. As a consequence, the regular initialisation routine is aborted in its entirety, by carrying out step 175.

Fig. 18 shows the enquiry made by the loading machine (one of M1, M2...Mn) to the server machine X of Fig. 15. The operation of the loading machine is temporarily interrupted as indicated by step 181 until the reply is received from machine X, as indicated by step 182.

Fig. 19 shows the activity carried out by machine X of Fig. 15 in response to such an enquiry as step 181 of Fig. 18. The initialisation status is determined in steps 192 and 193 and, if already initialised, the response to that effect is sent to the enquiring machine by carrying out step 194. Similarly, if the initialisation status is uninitialized, the corresponding reply is sent by carrying out steps 195 and 196. The waiting enquiring machine created by step 182 is then able to respond accordingly.

Reference is made to the accompanying Annexures in which:
Annexures A1-A10 illustrate actual code in relation to fields,
Annexure B1 is a typical code fragment from an unmodified <clinit> instruction,
Annexure B2 is an equivalent in respect of a modified <clinit> instruction,
Annexure B3 is a typical code fragment from an unmodified <init> instruction,
Annexure B4 is an equivalent in respect of a modified <init> instruction,
In addition, Annexure B5 is an alternative to the code of Annexure B2,
Annexure B6 is an alternative to the code of Annexure B4.
Furthermore, Annexure B7 is the source-code of InitClient, which queries an "initialization server" for the initialization status of the relevant class or object.

Annexure B8 is the source-code of InitServer, which receives an initialization status query by InitClient and in response returns the corresponding status.

Similarly, Annexure B9 is the source-code of the example application used in the before/after examples of Annexure B1-B6.

Turning now to Fig. 20, the procedure followed to modify the <clinit> routine relating to classes so as to convert from the code fragment of Annexure B1 to the code fragment of Annexure B2 is indicated. The initial loading of the application program 50 onto the JAVA virtual machine 72 is commenced at step 201, and each line of code is scrutinized in order to detect those instructions which represent the <clinit> routine by carrying out step 202. Once so detected, the <clinit> routine is modified as indicated in Annexure B2 by carrying out step 203. As indicated by step 204, after the modification is completed the loading procedure is then continued.

Annexures B1 and B2 are the before and after excerpt of a <clinit> instruction respectively. The modified code that is added to the method is highlighted in bold. In the original code sample of Annexure B1, the <clinit> method creates a new object of itself, and writes this to the memory location (field) called "thisTest". Thus, without management of class loading in a distributed environment, each machine would re-initialise the same shared memory location (field), with different objects. Clearly this is not what the programmer of the application program being loaded expects to happen.

So, taking advantage of the DRT, the application code is modified as it is loaded into the machine by changing the <clinit> method. The changes made (highlighted in bold) are the initial instructions that the <clinit> method executes. These added instructions check if this class has already been loaded by calling the isAlreadyLoaded() method, which returns either true or false corresponding to the loaded state of this class.

The isAlreadyLoaded() method of the DRT can optionally take an argument which represents a unique identifier for this class (See ANNEXURE B5 and B6), for example the name of the class, or a class object representing this class, or a unique

number representing this class across all machines, to be used in the determination of the loaded status of this class. This way, the DRT can support the loading of multiple classes at the same time without becoming confused as to which of the multiple classes are already loaded and which are not, by using the unique identifier of each class to consult the correct record in the isAlreadyLoaded table.

The DRT can determine the loaded state of the class in a number of ways. Preferably, it can ask each machine in turn if this class is loaded, and if any machine replies true, then return true, otherwise false. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on the local machine, or a database) to determine if this class has been loaded or not.

If the DRT returns false, then this means that this class has not been loaded before on any machine in the distributed environment, and hence, this execution is to be considered the first and original. As a result, the DRT must update the "isAlreadyLoaded" record for this class in the shared record table to true, such that all subsequent invocations of isAlreadyLoaded on all other machines, and including the current machine, will now return true. Thus, if DRT.isAlreadyLoaded() returns false, the modified <clinit> method proceeds with the original code block, which now trails the inserted three instructions.

On the other hand, if the DRT returns true, then this means that this class has already been loaded in the distributed environment, as recorded in the shared record table of loaded classes. In such a case, the original code block is **NOT** to be executed, as it will overwrite already-initialised memory locations etc. Thus, when the DRT returns true, the inserted three instructions prevent execution of the original code, and return straight away to the application program.

An equivalent procedure for the <init> routines relating to objects is illustrated in Fig. 21 where steps 212 and 213 are equivalent to steps 202 and 203 of Fig. 20. This results in the code of Annexure B3 being converted into the code of Annexure B4.

A similar modification as used for <clinit> is used for <init>. The application program's <init> block (or blocks, as there can be multiple - unlike <clinit>) is or are detected as shown by step 212 and modified as shown by step 213 to behave coherently across the distributed environment.

In the example of Annexure B3 the application program's <init> instructions initialise a memory location (field) with the timestamp of the loading time. The application could use this, for example, to record when this object was created. Clearly, in a distributed environment, where peer objects can load at different times, special treatment is necessary to make sure that the timestamp of the first-loaded peer object is not overwritten by later peer objects.

The disassembled instruction sequence after modification has taken place is set out in Annexure B4 and the modified/inserted instructions are highlighted in bold. For the <init> modification, unlike the <clinit> modification, the modifying instructions are often required to be placed after the "invokespecial" instruction, instead of at the very beginning. The reasons for this are driven by the JAVA Virtual Machine specification. Other languages often have similar subtle designs nuances.

Given the fundamental concept of testing to see if initialization has already been carried out, and if not carrying it out, and if so, not carrying out any further initialization; there are several different ways in which this concept can be carried out.

In the first embodiment, a particular machine, say machine M2, loads the class or object on itself and then loads each of the other machines M1, M3 ... Mn (either sequentially or simultaneously). In this arrangement, which may be termed "master/slave" each of machines M1, M3, ... Mn loads what it is given by machine M2.

In a variation of this "master/slave" arrangement, machine M2 loads the <clinit> routine in unmodified form on machine M2 and then modifies the class by deleting the initialization routine in its entirety and loads the modified class on the

other machines. Thus in this instance the modification is not a by-passing of the initialization routine but a deletion of it on all machines except one.

In a still further embodiment, each machine receives the initialization routine, but modifies it and loads the modified routine on that machine. This enables the modification carried out by each machine to be slightly different being optimized based upon its architecture and operating system, yet still coherent with all other similar modifications..

In a further arrangement, a particular machine, say M1, loads the class and all other machines M2, M3 … Mn do a modification to delete the initialization routine and load the modified version.

In all instances, the supply can be branched (ie M2 supplies each of M1, M3, M4, etc directly) or cascaded or sequential (ie M2 applies M1 which then supplies M3 which then supplies M4, and so on).

In a still further arrangement, the initial machine, say M2, can carry out the initial loading and then generate a table which lists all the classes loaded by machine M2. This table is then sent to all other machines (either in branched or cascade fashion). Then if a machine, other than M2, needs to access a class listed in the table, it sends a request to M2 to provide the necessary information. Thus the information provided to machine Mn is, in general, different from the initial state loaded into machine M2.

Under the above circumstances it is necessary for each entry in the table to be accompanied by a counter which is incremented on each occasion that a class is loaded. Thus, when data is demanded, both the class contents and the count of the corresponding counter are transferred in response to the demand. This “on demand” mode increases the overhead of each computer but reduces the volume of traffic on the communications network which interconnects the computers.

In a still further arrangement, the machines M1 to Mn, can send all load requests to an additional machine X (of Fig. 15), which performs the modification via any of the afore mentioned methods, and returns the modified class to each of the machines M1 to Mn which then load the class locally. In this arrangement, machines M1 to Mn do not maintain a table of records for any class, and instead, they forward all load requests to machine X, which maintains the table of loaded classes, and returns a modified class to each machine dependant on whether or not it is the first time a given class is loaded on machines M1 to Mn. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention.

Persons skilled in the computing arts will be aware of four techniques used in creating modifications in computer code. The first is to make the modification in the original (source) language. The second is to convert the original code (in say JAVA) into an intermediate representation (or intermediate language). Once this conversion takes place the modification is made and then the conversion is reversed. This gives the desired result of modified JAVA code.

The third possibility is to convert to machine code (either directly or via the abovementioned intermediate language). Then the machine code is modified before being loaded and executed. The fourth possibility is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all four modification routes and also a combination of two, three or even all four, of such routes.

Turning now to Figs. 22-24, two laptop computers 101 and 102 are illustrated. The computers 101 and 102 are not necessarily identical and indeed, one can be an IBM or IBM-clone and the other can be an APPLE computer. The computers 101 and 102 have two screens 105, 115 two keyboards 106, 116 but a single mouse 107. The two machines 101, 102 are interconnected by a means of a single coaxial cable or twisted pair cable 314.

Two simple application programs are downloaded onto each of the machines 101, 102, the programs being modified as they are being loaded as described above. In this embodiment the first application is a simple calculator program and results in the image of a calculator 108 being displayed on the screen 105. The second program is a graphics program which displays four coloured blocks 109 which are of different colours and which move about at random within a rectangular box 310. Again, after loading, the box 310 is displayed on the screen 105. Each application operates independently so that the blocks 109 are in random motion on the screen 105 whilst numerals within the calculator 108 can be selected (with the mouse 107) together with a mathematical operator (such as addition or multiplication) so that the calculator 108 displays the result.

The mouse 107 can be used to "grab" the box 310 and move same to the right across the screen 105 and onto the screen 115 so as to arrive at the situation illustrated in Fig. 23. In this arrangement, the calculator application is being conducted on machine 101 whilst the graphics application resulting in display of box 310 is being conducted on machine 102.

However, as illustrated in Fig. 24, it is possible by means of the mouse 107 to drag the calculator 108 to the right as seen in Fig. 23 so as to have a part of the calculator 108 displayed by each of the screens 105, 115. Similarly, the box 310 can be dragged by means of the mouse 107 to the left as seen in Fig. 23 so that the box 310 is partially displayed by each of the screens 105, 115 as indicated Fig. 24. In this configuration, part of the calculator operation is being performed on machine 101 and part on machine 102 whilst part of the graphics application is being carried out the machine 101 and the remainder is carried out on machine 102.

The foregoing describes only some embodiments of the present invention and modifications, obvious to those skilled in the art, can be made thereto without departing from the scope of the present invention. For example, reference to JAVA includes both the JAVA language and also JAVA platform and architecture.

Those skilled in the programming arts will be aware that when additional code or instructions is/are inserted into an existing code or instruction set to modify same, the existing code or instruction set may well require further modification (eg by

re-numbering of sequential instructions) so that offsets, branching, attributes, mark up and the like are catered for.

Similarly, in the JAVA language memory locations include, for example, both fields and array types. The above description deals with fields and the changes required for array types are essentially the same mutatis mutandis. Also the present invention is equally applicable to similar programming languages (including procedural, declarative and object orientated) to JAVA including Micrsoft.NET platform and architecture (Visual Basic, Visual C/C^{++}, and C#) FORTRAN, C/C^{++}, COBOL, BASIC etc.

The abovementioned embodiment in which the code of the JAVA initialisation routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will call the JAVA initialisation routine. It is possible to leave the JAVA initialisation routine unamended and instead amend the LINUX or HOTSPOT routine which calls the JAVA initialisation routine, so that if the object or class is already loaded, then the JAVA initialisation routine is not called. In order to embrace such an arrangement the term "initialisation routine" is to be understood to include within its scope both the JAVA initialisation routine and the "combination" of the JAVA initialisation routine <u>and</u> the LINUX or HOTSPOT code fragments which call or initiates the JAVA initialisation routine.

The terms object and class used herein are derived from the JAVA environment and are intended to embrace similar terms derived from different environments such as dynamically linked libraries (DLL), or object code packages, or function unit or memory locations.

The term "comprising" (and its grammatical variations) as used herein is used in the inclusive sense of "having" or "including" and not in the exclusive sense of "consisting only of".

Annexure A

The following are program listings in the JAVA language:

A1. This first excerpt is part of the modification code. It searches through the code array, and when it finds a putstatic instruction (opcode 178), it implements the modifications.

```
// START
byte[] code =  Code_attribute.code;      // Bytecode of a given method in a
                                         // given classfile.

int code_length = Code_attribute.code_length;

int DRT = 99;          // Location of the CONSTANT_Methodref_info for the
                       // DRT.alert() method.

for (int i=0; i<code_length; i++){

   if ((code[i] & 0xff) == 179){  // Putstatic instruction.

      System.arraycopy(code, i+3, code, i+6, code_length-(i+3));

      code[i+3] = (byte) 184;     // Invokestatic instruction for the
                                  // DRT.alert() method.

      code[i+4] = (byte) ((DRT >>> 8) & 0xff);

      code[i+5] = (byte) (DRT & 0xff);

   }

}
// END
```

A2. This second excerpt is part of the DRT.alert() method. This is the body of the DRT.alert() method when it is called.

```
// START
public static void alert(){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.notify(); // Alerts a waiting DRT thread in the background.

   }

}
// END
```

A3. This third excerpt is part of the DRT Sending. This code fragment shows the DRT in a separate thread, after being notified, sending the value across the network.

```
// START
MulticastSocket ms = DRT.getMulticastSocket();        // The multicast socket
                                                      // used by the DRT for
                                                      // communication.


byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.
```

```
Field field = modifiedClass.getDeclaredField("myField1");      // Stores
                                                               // the field
                                                               // from the
                                                               // modified
                                                               // class.

// In this example, the field is a byte field.
while (DRT.isRunning()){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.wait();    // The DRT thread is waiting for the alert
                            // method to be called.

      byte[] b = new byte[]{nameTag, field.getByte(null)};  // Stores
                                                            // the
                                                            // nameTag
                                                            // and the
                                                            // value
                                                            // of the
                                                            // field from
                                                            // the modified
                                                            // class in a
                                                            // buffer.

      DatagramPacket dp = new DatagramPacket(b, 0, b.length);

      ms.send(dp);   // Send the buffer out across the network.

   }

}
// END
```

A4. The fourth excerpt is part of the DRT receiving. This is a fragment of code to receive a DRT sent alert over the network.

```
// START
MulticastSocket ms = DRT.getMulticastSocket(); // The multicast socket
                                               // used by the DRT for
                                               // communication.

DatagramPacket dp = new DatagramPacket(new byte[2], 0, 2);

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");     // Stores the
                                                              // field from
                                                              // the modified
                                                              // class.


// In this example, the field is a byte field.
while (DRT.isRunning){

   ms.receive(dp);   // Receive the previously sent buffer from the network.

   byte[] b = dp.getData();

   if (b[0] == nameTag){   // Check the nametags match.
      field.setByte(null, b[1]);  // Write the value from the network packet
                                  // into the field location in memory.

   }
```

```
}
// END
```

A5. The fifth excerpt is an example application before modification has occurred.
Method void setValues(int, int)
0 iload_1
1 putstatic #3 <Field int staticValue>
4 aload_0
5 iload_2
6 putfield #2 <Field int instanceValue>
9 return

A6. The sixth excerpt is the same example application in 5 after modification has been performed. The modifications are highlighted in **bold.**
Method void setValues(int, int)
0 iload_1
1 putstatic #3 <Field int staticValue>
4 ldc #4 <String "example">
6 iconst_0
7 invokestatic #5 <Method void alert(java.lang.Object, int)>
10 aload_0
11 iload_2
12 putfield #2 <Field int instanceValue>
15 aload_0
16 iconst_1
17 invokestatic #5 <Method void alert(java.lang.Object, int)>
20 return

A7. The seventh excerpt is the source-code of the example application used in excerpt 5 and 6.

```
import java.lang.*;

public class example{


   /** Shared static field. */
   public static int staticValue = 0;


   /** Shared instance field. */
   public int instanceValue = 0;


   /** Example method that writes to memory (instance field). */
   public void setValues(int a, int b){

      staticValue = a;

      instanceValue = b;

   }

}
```

A8. The eighth excerpt is the source-code of FieldAlert, which alerts the "distributed run-time" to propagate a changed value.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FieldAlert{


   /** Table of alerts. */
   public final static Hashtable alerts = new Hashtable();


   /** Object handle. */
   public Object reference = null;


   /** Table of field alerts for this object. */
   public boolean[] fieldAlerts = null;


   /** Constructor. */
   public FieldAlert(Object o, int initialFieldCount){
      reference = o;
      fieldAlerts = new boolean[initialFieldCount];
   }


   /** Called when an application modifies a value. (Both objects and
       classes) */
   public static void alert(Object o, int fieldID){

      // Lock the alerts table.
      synchronized (alerts){

         FieldAlert alert = (FieldAlert) alerts.get(o);

         if (alert == null){      // This object hasn't been alerted already,
                                  // so add to alerts table.
            alert = new FieldAlert(o, fieldID + 1);
            alerts.put(o, alert);
         }

         if (fieldID >= alert.fieldAlerts.length){
            // Ok, enlarge fieldAlerts array.
            boolean[] b = new boolean[fieldID+1];
            System.arraycopy(alert.fieldAlerts, 0, b, 0,
               alert.fieldAlerts.length);
            alert.fieldAlerts = b;
         }

         // Record the alert.
         alert.fieldAlerts[fieldID] = true;

         // Mark as pending.
         FieldSend.pending = true;      // Signal that there is one or more
                                        // propagations waiting.

         // Finally, notify the waiting FieldSend thread(s)
         if (FieldSend.waiting){
            FieldSend.waiting = false;
            alerts.notify();
```

```
            }

        }

    }

}
```

A9. The ninth excerpt is the source-code of FieldSend, which propagates changes values alerted to it via FieldAlert.

```
import java.lang.*;
import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FieldSend implements Runnable{


   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int PROPAGATE_OBJECT = 10;
   public final static int PROPAGATE_CLASS = 20;


   /** FieldAlert network values. */
   public final static String group =
      System.getProperty("FieldAlert_network_group");
   public final static int port =
      Integer.parseInt(System.getProperty("FieldAlert_network_port"));


   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable objectToGlobalID = new Hashtable();


   /** Table of global ID's for local classnames. (classname-to-globalID
       mappings) */
   public final static Hashtable classNameToGlobalID = new Hashtable();


   /** Pending. True if a propagation is pending. */
   public static boolean pending = false;


   /** Waiting. True if the FieldSend thread(s) are waiting. */
   public static boolean waiting = false;


   /** Background send thread. Propagates values as this thread is alerted
       to their alteration. */
   public void run(){

      System.out.println("FieldAlert_network_group=" + group);
      System.out.println("FieldAlert_network_port=" + port);

      try{

         // Create a DatagramSocket to send propagated field values.
```

```
DatagramSocket datagramSocket =
   new DatagramSocket(port, InetAddress.getByName(group));

// Next, create the buffer and packet for all transmissions.
byte[] buffer = new byte[512];        // Working limit of 512 bytes
                                      // per packet.
DatagramPacket datagramPacket =
   new DatagramPacket(buffer, 0, buffer.length);

while (!Thread.interrupted()){

   Object[] entries = null;

   // Lock the alerts table.
   synchronized (FieldAlert.alerts){

      // Await for an alert to propagate something.
      while (!pending){
         waiting = true;
         FieldAlert.alerts.wait();
         waiting = false;
      }

      pending = false;

      entries = FieldAlert.alerts.entrySet().toArray();

      // Clear alerts once we have copied them.
      FieldAlert.alerts.clear();

   }

   // Process each object alert in turn.
   for (int i=0; i<entries.length; i++){

      FieldAlert alert = (FieldAlert) entries[i];

      int index = 0;
      datagramPacket.setLength(buffer.length);

      Object reference = null;
      if (alert.reference instanceof String){
         // PROPAGATE_CLASS field operation.

         buffer[index++] = (byte) ((PROPAGATE_CLASS >> 24) & 0xff);
         buffer[index++] = (byte) ((PROPAGATE_CLASS >> 16) & 0xff);
         buffer[index++] = (byte) ((PROPAGATE_CLASS >> 8) & 0xff);
         buffer[index++] = (byte) ((PROPAGATE_CLASS >> 0) & 0xff);

         String name = (String) alert.reference;
         int length = name.length();
         buffer[index++] = (byte) ((length >> 24) & 0xff);
         buffer[index++] = (byte) ((length >> 16) & 0xff);
         buffer[index++] = (byte) ((length >> 8) & 0xff);
         buffer[index++] = (byte) ((length >> 0) & 0xff);

         byte[] bytes = name.getBytes();
         System.arraycopy(bytes, 0, buffer, index, length);
         index += length;

      }else{                    // PROPAGATE_OBJECT field operation.

         buffer[index++] =
            (byte) ((PROPAGATE_OBJECT >> 24) & 0xff);
         buffer[index++] =
            (byte) ((PROPAGATE_OBJECT >> 16) & 0xff);
```

```
         buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 8) & 0xff);
         buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 0) & 0xff);

         int globalID = ((Integer)
            objectToGlobalID.get(alert.reference)).intValue();

         buffer[index++] = (byte) ((globalID >> 24) & 0xff);
         buffer[index++] = (byte) ((globalID >> 16) & 0xff);
         buffer[index++] = (byte) ((globalID >> 8) & 0xff);
         buffer[index++] = (byte) ((globalID >> 0) & 0xff);

         reference = alert.reference;

      }

      // Use reflection to get a table of fields that correspond to
      // the field indexes used internally.
      Field[] fields = null;
      if (reference == null){
         fields = FieldLoader.loadClass((String)
            alert.reference).getDeclaredFields();
      }else{
         fields = alert.reference.getClass().getDeclaredFields();
      }

      // Now encode in batch mode the fieldID/value pairs.
      for (int j=0; j<alert.fieldAlerts.length; j++){

         if (alert.fieldAlerts[j] == false)
            continue;

         buffer[index++] = (byte) ((j >> 24) & 0xff);
         buffer[index++] = (byte) ((j >> 16) & 0xff);
         buffer[index++] = (byte) ((j >> 8) & 0xff);
         buffer[index++] = (byte) ((j >> 0) & 0xff);

         // Encode value.
         Class type = fields[j].getType();
         if (type == Boolean.TYPE){
            buffer[index++] =(byte)
               (fields[j].getBoolean(reference)? 1 : 0);
         }else if (type == Byte.TYPE){
            buffer[index++] = fields[j].getByte(reference);
         }else if (type == Short.TYPE){
            short v = fields[j].getShort(reference);
            buffer[index++] = (byte) ((v >> 8) & 0xff);
            buffer[index++] = (byte) ((v >> 0) & 0xff);
         }else if (type == Character.TYPE){
            char v = fields[j].getChar(reference);
            buffer[index++] = (byte) ((v >> 8) & 0xff);
            buffer[index++] = (byte) ((v >> 0) & 0xff);
         }else if (type == Integer.TYPE){
            int v = fields[j].getInt(reference);
            buffer[index++] = (byte) ((v >> 24) & 0xff);
            buffer[index++] = (byte) ((v >> 16) & 0xff);
            buffer[index++] = (byte) ((v >> 8) & 0xff);
            buffer[index++] = (byte) ((v >> 0) & 0xff);
         }else if (type == Float.TYPE){
            int v = Float.floatToIntBits(
               fields[j].getFloat(reference));
            buffer[index++] = (byte) ((v >> 24) & 0xff);
            buffer[index++] = (byte) ((v >> 16) & 0xff);
            buffer[index++] = (byte) ((v >> 8) & 0xff);
            buffer[index++] = (byte) ((v >> 0) & 0xff);
         }else if (type == Long.TYPE){
            long v = fields[j].getLong(reference);
```

```
               buffer[index++] = (byte) ((v >> 56) & 0xff);
               buffer[index++] = (byte) ((v >> 48) & 0xff);
               buffer[index++] = (byte) ((v >> 40) & 0xff);
               buffer[index++] = (byte) ((v >> 32) & 0xff);
               buffer[index++] = (byte) ((v >> 24) & 0xff);
               buffer[index++] = (byte) ((v >> 16) & 0xff);
               buffer[index++] = (byte) ((v >> 8) & 0xff);
               buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Double.TYPE){
               long v = Double.doubleToLongBits(
                  fields[j].getDouble(reference));
               buffer[index++] = (byte) ((v >> 56) & 0xff);
               buffer[index++] = (byte) ((v >> 48) & 0xff);
               buffer[index++] = (byte) ((v >> 40) & 0xff);
               buffer[index++] = (byte) ((v >> 32) & 0xff);
               buffer[index++] = (byte) ((v >> 24) & 0xff);
               buffer[index++] = (byte) ((v >> 16) & 0xff);
               buffer[index++] = (byte) ((v >> 8) & 0xff);
               buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else{
               throw new AssertionError("Unsupported type.");
            }


         }

         // Now set the length of the datagrampacket.
         datagramPacket.setLength(index);

         // Now send the packet.
         datagramSocket.send(datagramPacket);

      }

   }

   }catch (Exception e){
      throw new AssertionError("Exception: " + e.toString());
   }

}


}
```

A10. The tenth excerpt is the source-code of FieldReceive, which receives propagated changed values sent via FieldSend.

```
import java.lang.*;
import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FieldReceive implements Runnable{


   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int PROPAGATE_OBJECT = 10;
   public final static int PROPAGATE_CLASS = 20;
```

```
/** FieldAlert network values. */
public final static String group =
   System.getProperty("FieldAlert_network_group");
public final static int port =
   Integer.parseInt(System.getProperty("FieldAlert_network_port"));


/** Table of global ID's for local objects. (globalID-to-hashcode
    mappings) */
public final static Hashtable globalIDToObject = new Hashtable();


/** Table of global ID's for local classnames. (globalID-to-classname
    mappings) */
public final static Hashtable globalIDToClassName = new Hashtable();


/** Called when an application is to acquire a lock. */
public void run(){

   System.out.println("FieldAlert_network_group=" + group);
   System.out.println("FieldAlert_network_port=" + port);

   try{

      // Create a DatagramSocket to send propagated field values from
      MulticastSocket multicastSocket = new MulticastSocket(port);
      multicastSocket.joinGroup(InetAddress.getByName(group));

      // Next, create the buffer and packet for all transmissions.
      byte[] buffer = new byte[512];              // Working limit of 512
                                                  // bytes per packet.
      DatagramPacket datagramPacket =
         new DatagramPacket(buffer, 0, buffer.length);

      while (!Thread.interrupted()){

         // Make sure to reset length.
         datagramPacket.setLength(buffer.length);

         // Receive the next available packet.
         multicastSocket.receive(datagramPacket);

         int index = 0, length = datagramPacket.getLength();

         // Decode the command.
         int command = (int) (((buffer[index++] & 0xff) << 24)
            | ((buffer[index++] & 0xff) << 16)
            | ((buffer[index++] & 0xff) << 8)
            | (buffer[index++] & 0xff));

         if (command == PROPAGATE_OBJECT){  // Propagate operation for
                                            // object fields.

            // Decode global id.
            int globalID = (int) (((buffer[index++] & 0xff) << 24)
               | ((buffer[index++] & 0xff) << 16)
               | ((buffer[index++] & 0xff) << 8)
               | (buffer[index++] & 0xff));

            // Now, need to resolve the object in question.
            Object reference = globalIDToObject.get(
               new Integer(globalID));

            // Next, get the array of fields for this object.
            Field[] fields = reference.getClass().getDeclaredFields();
```

```
    while (index < length){

       // Decode the field id.
       int fieldID = (int) (((buffer[index++] & 0xff) << 24)
          | ((buffer[index++] & 0xff) << 16)
          | ((buffer[index++] & 0xff) << 8)
          | (buffer[index++] & 0xff));

       // Determine value length based on corresponding field
       // type.
       Field field = fields[fieldID];
       Class type = field.getType();
       if (type == Boolean.TYPE){
          boolean v = (buffer[index++] == 1 ? true : false);
          field.setBoolean(reference, v);
       }else if (type == Byte.TYPE){
          byte v = buffer[index++];
          field.setByte(reference, v);
       }else if (type == Short.TYPE){
          short v = (short) (((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setShort(reference, v);
       }else if (type == Character.TYPE){
          char v = (char) (((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setChar(reference, v);
       }else if (type == Integer.TYPE){
          int v = (int) (((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setInt(reference, v);
       }else if (type == Float.TYPE){
          int v = (int) (((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setFloat(reference, Float.intBitsToFloat(v));
       }else if (type == Long.TYPE){
          long v = (long) (((buffer[index++] & 0xff) << 56)
             | ((buffer[index++] & 0xff) << 48)
             | ((buffer[index++] & 0xff) << 40)
             | ((buffer[index++] & 0xff) << 32)
             | ((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setLong(reference, v);
       }else if (type == Double.TYPE){
          long v = (long) (((buffer[index++] & 0xff) << 56)
             | ((buffer[index++] & 0xff) << 48)
             | ((buffer[index++] & 0xff) << 40)
             | ((buffer[index++] & 0xff) << 32)
             | ((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setDouble(reference, Double.longBitsToDouble(v));
       }else{
          throw new AssertionError("Unsupported type.");
       }

    }

 }else if (command == PROPAGATE_CLASS){    // Propagate an update
```

```
                                     // to class fields.

    // Decode the classname.
    int nameLength = (int) (((buffer[index++] & 0xff) << 24)
       | ((buffer[index++] & 0xff) << 16)
       | ((buffer[index++] & 0xff) << 8)
       | (buffer[index++] & 0xff));
    String name = new String(buffer, index, nameLength);
    index += nameLength;

    // Next, get the array of fields for this class.
    Field[] fields =
       FieldLoader.loadClass(name).getDeclaredFields();

    // Decode all batched fields included in this propagation
    // packet.
    while (index < length){

       // Decode the field id.
       int fieldID = (int) (((buffer[index++] & 0xff) << 24)
          | ((buffer[index++] & 0xff) << 16)
          | ((buffer[index++] & 0xff) << 8)
          | (buffer[index++] & 0xff));

       // Determine field type to determine value length.
       Field field = fields[fieldID];
       Class type = field.getType();
       if (type == Boolean.TYPE){
          boolean v = (buffer[index++] == 1 ? true : false);
          field.setBoolean(null, v);
       }else if (type == Byte.TYPE){
          byte v = buffer[index++];
          field.setByte(null, v);
       }else if (type == Short.TYPE){
          short v = (short) (((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setShort(null, v);
       }else if (type == Character.TYPE){
          char v = (char) (((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setChar(null, v);
       }else if (type == Integer.TYPE){
          int v = (int) (((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setInt(null, v);
       }else if (type == Float.TYPE){
          int v = (int) (((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setFloat(null, Float.intBitsToFloat(v));
       }else if (type == Long.TYPE){
          long v = (long) (((buffer[index++] & 0xff) << 56)
             | ((buffer[index++] & 0xff) << 48)
             | ((buffer[index++] & 0xff) << 40)
             | ((buffer[index++] & 0xff) << 32)
             | ((buffer[index++] & 0xff) << 24)
             | ((buffer[index++] & 0xff) << 16)
             | ((buffer[index++] & 0xff) << 8)
             | (buffer[index++] & 0xff));
          field.setLong(null, v);
       }else if (type == Double.TYPE){
          long v = (long) (((buffer[index++] & 0xff) << 56)
             | ((buffer[index++] & 0xff) << 48)
```

```
                        | ((buffer[index++] & 0xff) << 40)
                        | ((buffer[index++] & 0xff) << 32)
                        | ((buffer[index++] & 0xff) << 24)
                        | ((buffer[index++] & 0xff) << 16)
                        | ((buffer[index++] & 0xff) << 8)
                        | (buffer[index++] & 0xff));
                     field.setDouble(null, Double.longBitsToDouble(v));
                  }else{          // Unsupported field type.
                     throw new AssertionError("Unsupported type.");
                  }

               }

            }

         }

      }catch (Exception e){
         throw new AssertionError("Exception: " + e.toString());
      }

   }


}
```

A11. FieldLoader.java

This excerpt is the source-code of FieldLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class FieldLoader extends URLClassLoader{


   public FieldLoader(URL[] urls){
      super(urls);
   }


   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{

         BufferedInputStream in =
            new BufferedInputStream(findResource(
            name.replace('.', '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      // Class-wide pointers to the ldc and alert index.
      int ldcindex = -1;
      int alertindex = -1;

      for (int i=0; i<cf.methods_count; i++){
         for (int j=0; j<cf.methods[i].attributes_count; j++){
```

```
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute)
cf.methods[i].attributes[j];

            boolean changed = false;

            for (int z=0; z<ca.code.length; z++){
               if ((ca.code[z][0] & 0xff) == 179){    // Opcode for a
PUTSTATIC
                                                      // instruction.
                  changed = true;

                  // The code below only supports fields in this class.
                  // Thus, first off, check that this field is local to this
                  // class.
                  CONSTANT_Fieldref_info fi = (CONSTANT_Fieldref_info)
                     cf.constant_pool[(int) (((ca.code[z][1] & 0xff) << 8) |
                     (ca.code[z][2] & 0xff))];
                  CONSTANT_Class_info ci = (CONSTANT_Class_info)
                     cf.constant_pool[fi.class_index];
                  String className =
                     cf.constant_pool[ci.name_index].toString();
                  if (!name.equals(className)){
                     throw new AssertionError("This code only supports
fields "
                        "local to this class");
                  }


                  // Ok, now search for the fields name and index.
                  int index = 0;
                  CONSTANT_NameAndType_info ni = (CONSTANT_NameAndType_info)
                     cf.constant_pool[fi.name_and_type_index];
                  String fieldName =
                     cf.constant_pool[ni.name_index].toString();
                  for (int a=0; a<cf.fields_count; a++){
                     String fn = cf.constant_pool[
                        cf.fields[a].name_index].toString();
                     if (fieldName.equals(fn)){
                        index = a;
                        break;
                     }
                  }


                  // Next, realign the code array, making room for the
                  // insertions.
                  byte[][] code2 = new byte[ca.code.length+3][];
                  System.arraycopy(ca.code, 0, code2, 0, z+1);
                  System.arraycopy(ca.code, z+1, code2, z+4,
                     ca.code.length-(z+1));
                  ca.code = code2;


                  // Next, insert the LDC_W instruction.
                  if (ldcindex == -1){
                     CONSTANT_String_info csi =
                        new CONSTANT_String_info(ci.name_index);
                     cp_info[] cpi = new cp_info[cf.constant_pool.length+1];
                     System.arraycopy(cf.constant_pool, 0, cpi, 0,
                        cf.constant_pool.length);
                     cpi[cpi.length - 1] = csi;
                     ldcindex = cpi.length-1;
                     cf.constant_pool = cpi;
```

```
                    cf.constant_pool_count++;
                }
                ca.code[z+1] = new byte[3];
                ca.code[z+1][0] = (byte) 19;
                ca.code[z+1][1] = (byte) ((ldcindex >> 8) & 0xff);
                ca.code[z+1][2] = (byte) (ldcindex & 0xff);


                // Next, insert the SIPUSH instruction.
                ca.code[z+2] = new byte[3];
                ca.code[z+2][0] = (byte) 17;
                ca.code[z+2][1] = (byte) ((index >> 8) & 0xff);
                ca.code[z+2][2] = (byte) (index & 0xff);

                // Finally, insert the INVOKESTATIC instruction.
                if (alertindex == -1){
                    // This is the first time this class is encourtering
the
                    // alert instruction, so have to add it to the constant
                    // pool.
                    cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
                    System.arraycopy(cf.constant_pool, 0, cpi, 0,
                        cf.constant_pool.length);
                    cf.constant_pool = cpi;
                    cf.constant_pool_count += 6;

                    CONSTANT_Utf8_info u1 =
                        new CONSTANT_Utf8_info("FieldAlert");
                    cf.constant_pool[cf.constant_pool.length-6] = u1;

                    CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                        cf.constant_pool_count-6);
                    cf.constant_pool[cf.constant_pool.length-5] = c1;

                    u1 = new CONSTANT_Utf8_info("alert");
                    cf.constant_pool[cf.constant_pool.length-4] = u1;

                    u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;I)V");
                    cf.constant_pool[cf.constant_pool.length-3] = u1;

                    CONSTANT_NameAndType_info n1 =
                        new CONSTANT_NameAndType_info(
                        cf.constant_pool.length-4, cf.constant_pool.length-
3);
                    cf.constant_pool[cf.constant_pool.length-2] = n1;

                    CONSTANT_Methodref_info m1 = new
CONSTANT_Methodref_info(
                        cf.constant_pool.length-5, cf.constant_pool.length-
2);
                    cf.constant_pool[cf.constant_pool.length-1] = m1;
                    alertindex = cf.constant_pool.length-1;
                }
                ca.code[z+3] = new byte[3];
                ca.code[z+3][0] = (byte) 184;
                ca.code[z+3][1] = (byte) ((alertindex >> 8) & 0xff);
                ca.code[z+3][2] = (byte) (alertindex & 0xff);


                // And lastly, increase the CODE_LENGTH and
ATTRIBUTE_LENGTH
                // values.
                ca.code_length += 9;
                ca.attribute_length += 9;

            }
```

```
            }

            // If we changed this method, then increase the stack size by
one.
            if (changed){
               ca.max_stack++;          // Just to make sure.
            }

         }
      }


      try{

         ByteArrayOutputStream out = new ByteArrayOutputStream();

         cf.serialize(out);

         byte[] b = out.toByteArray();

         return defineClass(name, b, 0, b.length);

      }catch (Exception e){
         throw new ClassNotFoundException(name);
      }

   }

}
```

A12. **Attribute_info.java**

Convience class for representing attribute_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This abstract class represents all types of attribute_info
 *  that are used in the JVM specifications.
 *
 *  All new attribute_info subclasses are to always inherit from this
 *  class.
 */
public abstract class attribute_info{

    public int attribute_name_index;
    public int attribute_length;

    /** This is used by subclasses to register themselves
     *  to their parent classFile.
     */
    attribute_info(ClassFile cf){}

    /** Used during input serialization by ClassFile only. */
    attribute_info(ClassFile cf, DataInputStream in)
        throws IOException{
        attribute_name_index = in.readChar();
        attribute_length = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(attribute_name_index);
        out.writeInt(attribute_length);
    }
```

```
    /** This class represents an unknown attribute_info that
     *  this current version of classfile specification does
     *  not understand.
     */
    public final static class Unknown extends attribute_info{

        byte[] info;

        /** Used during input serialization by ClassFile only. */
        Unknown(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            info = new byte[attribute_length];
            in.read(info, 0, attribute_length);
        }

        /** Used during output serialization by ClassFile only. */
        void serialize(DataOutputStream out)
            throws IOException{
            ByteArrayOutputStream baos = new ByteArrayOutputStream();

            super.serialize(out);
            out.write(info, 0, attribute_length);
        }
    }

}
```

A13. ClassFile.java

Convience class for representing ClassFile structures.

```
import java.lang.*;
import java.io.*;
import java.util.*;

/** The ClassFile follows verbatim from the JVM specification.  */
public final class ClassFile {

    public int magic;
    public int minor_version;
    public int major_version;
    public int constant_pool_count;
    public cp_info[] constant_pool;
    public int access_flags;
    public int this_class;
    public int super_class;
    public int interfaces_count;
    public int[] interfaces;
    public int fields_count;
    public field_info[] fields;
    public int methods_count;
    public method_info[] methods;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Constructor. Takes in a byte stream representation and transforms
     *  each of the attributes in the ClassFile into objects to allow for
     *  easier manipulation.
     */
    public ClassFile(InputStream ins)
        throws IOException{
        DataInputStream in = (ins instanceof DataInputStream ?
            (DataInputStream) ins : new DataInputStream(ins));
        magic = in.readInt();
        minor_version = in.readChar();
```

```
        major_version = in.readChar();
        constant_pool_count = in.readChar();
        constant_pool = new cp_info[constant_pool_count];
        for (int i=1; i<constant_pool_count; i++){
            in.mark(1);
            int s = in.read();
            in.reset();
            switch (s){
                case 1:
                    constant_pool[i] = new CONSTANT_Utf8_info(this, in);
                    break;
                case 3:
                    constant_pool[i] = new CONSTANT_Integer_info(this, in);
                    break;
                case 4:
                    constant_pool[i] = new CONSTANT_Float_info(this, in);
                    break;
                case 5:
                    constant_pool[i] = new CONSTANT_Long_info(this, in);
                    i++;
                    break;
                case 6:
                    constant_pool[i] = new CONSTANT_Double_info(this, in);
                    i++;
                    break;
                case 7:
                    constant_pool[i] = new CONSTANT_Class_info(this, in);
                    break;
                case 8:
                    constant_pool[i] = new CONSTANT_String_info(this, in);
                    break;
                case 9:
                    constant_pool[i] = new CONSTANT_Fieldref_info(this, in);
                    break;
                case 10:
                    constant_pool[i] = new CONSTANT_Methodref_info(this,
in);
                    break;
                case 11:
                    constant_pool[i] =
                        new CONSTANT_InterfaceMethodref_info(this, in);
                    break;
                case 12:
                    constant_pool[i] = new CONSTANT_NameAndType_info(this,
in);
                    break;
                default:
                    throw new ClassFormatError("Invalid ConstantPoolTag");
            }
        }
        access_flags = in.readChar();
        this_class = in.readChar();
        super_class = in.readChar();
        interfaces_count = in.readChar();
        interfaces = new int[interfaces_count];
        for (int i=0; i<interfaces_count; i++)
            interfaces[i] = in.readChar();
        fields_count = in.readChar();
        fields = new field_info[fields_count];
        for (int i=0; i<fields_count; i++) {
            fields[i] = new field_info(this, in);
        }
        methods_count = in.readChar();
        methods = new method_info[methods_count];
        for (int i=0; i<methods_count; i++) {
            methods[i] = new method_info(this, in);
```

```
        }
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("SourceFile"))
                attributes[i] = new SourceFile_attribute(this, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(this, in);
            else if (s.equals("InnerClasses"))
                attributes[i] = new InnerClasses_attribute(this, in);
            else
                attributes[i] = new attribute_info.Unknown(this, in);
        }
    }

    /** Serializes the ClassFile object into a byte stream. */
    public void serialize(OutputStream o)
        throws IOException{
        DataOutputStream out = (o instanceof DataOutputStream ?
            (DataOutputStream) o : new DataOutputStream(o));
        out.writeInt(magic);
        out.writeChar(minor_version);
        out.writeChar(major_version);
        out.writeChar(constant_pool_count);
        for (int i=1; i<constant_pool_count; i++){
            constant_pool[i].serialize(out);
            if (constant_pool[i] instanceof CONSTANT_Long_info ||
                    constant_pool[i] instanceof CONSTANT_Double_info)
                i++;
        }
        out.writeChar(access_flags);
        out.writeChar(this_class);
        out.writeChar(super_class);
        out.writeChar(interfaces_count);
        for (int i=0; i<interfaces_count; i++)
            out.writeChar(interfaces[i]);
        out.writeChar(fields_count);
        for (int i=0; i<fields_count; i++)
            fields[i].serialize(out);
        out.writeChar(methods_count);
        for (int i=0; i<methods_count; i++)
            methods[i].serialize(out);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
        // Flush the outputstream just to make sure.
        out.flush();
    }

}
```

A14. **Code_attribute.java**

Convience class for representing Code_attribute structures within ClassFiles.

```
import java.util.*;
import java.lang.*;
import java.io.*;

/**
 *  The code[] is stored as a 2D array.  */
public final class Code_attribute extends attribute_info{
```

```
    public int max_stack;
    public int max_locals;
    public int code_length;
    public byte[][] code;
    public int exception_table_length;
    public exception_table[] exception_table;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Internal class that handles the exception table. */
    public final static class exception_table{
        public int start_pc;
        public int end_pc;
        public int handler_pc;
        public int catch_type;
    }

    /** Constructor called only by method_info. */
    Code_attribute(ClassFile cf, int ani, int al, int ms, int ml, int cl,
                   byte[][] cd, int etl, exception_table[] et, int ac,
                   attribute_info[] a){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        max_stack = ms;
        max_locals = ml;
        code_length = cl;
        code = cd;
        exception_table_length = etl;
        exception_table = et;
        attributes_count = ac;
        attributes = a;
    }

    /** Used during input serialization by ClassFile only. */
    Code_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        max_stack = in.readChar();
        max_locals = in.readChar();
        code_length = in.readInt();
        code = new byte[code_length][];

        int i = 0;
        for (int pos=0; pos<code_length; i++){
            in.mark(1);
            int s = in.read();
            in.reset();
            switch (s){
                case 16:
                case 18:
                case 21:
                case 22:
                case 23:
                case 24:
                case 25:
                case 54:
                case 55:
                case 56:
                case 57:
                case 58:
                case 169:
                case 188:
                case 196:
                    code[i] = new byte[2];
                    break;
```

```
            case 17:
            case 19:
            case 20:
            case 132:
            case 153:
            case 154:
            case 155:
            case 156:
            case 157:
            case 158:
            case 159:
            case 160:
            case 161:
            case 162:
            case 163:
            case 164:
            case 165:
            case 166:
            case 167:
            case 168:
            case 178:
            case 179:
            case 180:
            case 181:
            case 182:
            case 183:
            case 184:
            case 187:
            case 189:
            case 192:
            case 193:
            case 198:
            case 199:
            case 209:
                code[i] = new byte[3];
                break;
            case 197:
                code[i] = new byte[4];
                break;
            case 185:
            case 200:
            case 201:
                code[i] = new byte[5];
                break;
            case 170:{
                int pad = 3 - (pos % 4);
                in.mark(pad+13);                            // highbyte
                in.skipBytes(pad+5);                        // lowbyte
                int low = in.readInt();
                code[i] =
                    new byte[pad + 13 + ((in.readInt() - low + 1) * 4)];
                in.reset();
                break;
            }case 171:{
                int pad = 3 - (pos % 4);
                in.mark(pad+9);
                in.skipBytes(pad+5);
                code[i] = new byte[pad + 9 + (in.readInt() * 8)];
                in.reset();
                break;
            }default:
                code[i] = new byte[1];
        }
        in.read(code[i], 0, code[i].length);
        pos += code[i].length;
    }
```

```
            // adjust the array to the new size and store the size
            byte[][] temp = new byte[i][];
            System.arraycopy(code, 0, temp, 0, i);
            code = temp;

            exception_table_length = in.readChar();
            exception_table =
                new Code_attribute.exception_table[exception_table_length];
            for (i=0; i<exception_table_length; i++){
                exception_table[i] = new exception_table();
                exception_table[i].start_pc = in.readChar();
                exception_table[i].end_pc = in.readChar();
                exception_table[i].handler_pc = in.readChar();
                exception_table[i].catch_type = in.readChar();
            }
            attributes_count = in.readChar();
            attributes = new attribute_info[attributes_count];
            for (i=0; i<attributes_count; i++){
                in.mark(2);
                String s = cf.constant_pool[in.readChar()].toString();
                in.reset();
                if (s.equals("LineNumberTable"))
                    attributes[i] = new LineNumberTable_attribute(cf, in);
                else if (s.equals("LocalVariableTable"))
                    attributes[i] = new LocalVariableTable_attribute(cf, in);
                else
                    attributes[i] = new attribute_info.Unknown(cf, in);
            }
        }

        /** Used during output serialization by ClassFile only.
        */
        void serialize(DataOutputStream out)
            throws IOException{
                attribute_length = 12 + code_length +
                    (exception_table_length * 8);
                for (int i=0; i<attributes_count; i++)
                    attribute_length += attributes[i].attribute_length + 6;
                super.serialize(out);
                out.writeChar(max_stack);
                out.writeChar(max_locals);
                out.writeInt(code_length);
                for (int i=0, pos=0; pos<code_length; i++){
                    out.write(code[i], 0, code[i].length);
                    pos += code[i].length;
                }
                out.writeChar(exception_table_length);
                for (int i=0; i<exception_table_length; i++){
                    out.writeChar(exception_table[i].start_pc);
                    out.writeChar(exception_table[i].end_pc);
                    out.writeChar(exception_table[i].handler_pc);
                    out.writeChar(exception_table[i].catch_type);
                }
                out.writeChar(attributes_count);
                for (int i=0; i<attributes_count; i++)
                    attributes[i].serialize(out);
        }

}
```

A15. **CONSTANT_Class_info.java**

Convience class for representing CONSTANT_Class_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Class subtype of a constant pool entry.  */
public final class CONSTANT_Class_info extends cp_info{

    /** The index to the name of this class. */
    public int name_index = 0;

    /** Convenience constructor.
     */
    public CONSTANT_Class_info(int index) {
        tag = 7;
        name_index = index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Class_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 7)
            throw new ClassFormatError();
        name_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
    }

}
```

A16. **CONSTANT_Double_info.java**

Convience class for representing CONSTANT_Double_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Double subtype of a constant pool entry. */
public final class CONSTANT_Double_info extends cp_info{

    /** The actual value. */
    public double bytes;

    public CONSTANT_Double_info(double d){
        tag = 6;
        bytes = d;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Double_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 6)
            throw new ClassFormatError();
        bytes = in.readDouble();
    }
```

```
    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeDouble(bytes);
        long l = Double.doubleToLongBits(bytes);
    }
}
```

A17. **CONSTANT_Fieldref_info.java**

Convience class for representing CONSTANT_Fieldref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Fieldref subtype of a constant pool entry.  */
public final class CONSTANT_Fieldref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    /** Convenience constructor. */
    public CONSTANT_Fieldref_info(int class_index, int name_and_type_index)
{
        tag = 9;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Fieldref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 9)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A18. **CONSTANT_Float_info.java**

Convience class for representing CONSTANT_Float_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Float subtype of a constant pool entry. */
```

```
public final class CONSTANT_Float_info extends cp_info{

    /** The actual value. */
    public float bytes;

    public CONSTANT_Float_info(float f){
        tag = 4;
        bytes = f;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Float_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 4)
            throw new ClassFormatError();
        bytes = in.readFloat();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(4);
        out.writeFloat(bytes);
    }

}
```

A19. CONSTANT_Integer_info.java

Convience class for representing CONSTANT_Integer_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Integer subtype of a constant pool entry.  */
public final class CONSTANT_Integer_info extends cp_info{

    /** The actual value. */
    public int bytes;

    public CONSTANT_Integer_info(int b) {
        tag = 3;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Integer_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 3)
            throw new ClassFormatError();
        bytes = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeInt(bytes);
    }
```

```
}
```

A20. **CONSTANT_InterfaceMethodref_info.java**
Convience class for representing CONSTANT_InterfaceMethodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** InterfaceMethodref subtype of a constant pool entry.
 */
public final class CONSTANT_InterfaceMethodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    public CONSTANT_InterfaceMethodref_info(int class_index,
                                            int name_and_type_index) {
        tag = 11;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_InterfaceMethodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 11)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A21. **CONSTANT_Long_info.java**
Convience class for representing CONSTANT_Long_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Long subtype of a constant pool entry.  */
public final class CONSTANT_Long_info extends cp_info{

    /** The actual value. */
    public long bytes;

    public CONSTANT_Long_info(long b){
```

```
        tag = 5;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Long_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 5)
            throw new ClassFormatError();
        bytes = in.readLong();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeLong(bytes);
    }

}
```

A22. **CONSTANT_Methodref_info.java**

Convience class for representing CONSTANT_Methodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Methodref subtype of a constant pool entry.
 */
public final class CONSTANT_Methodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    public CONSTANT_Methodref_info(int class_index, int name_and_type_index)
{
        tag = 10;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Methodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 10)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
```

```
        out.writeChar(name_and_type_index);
    }

}
```

A23. **CONSTANT_NameAndType_info.java**
Convience class for representing CONSTANT_NameAndType_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** NameAndType subtype of a constant pool entry.
 */
public final class CONSTANT_NameAndType_info extends cp_info{

    /** The index to the Utf8 that contains the name. */
    public int name_index;

    /** The index fo the Utf8 that constains the signature. */
    public int descriptor_index;

    public CONSTANT_NameAndType_info(int name_index, int descriptor_index) {
        tag = 12;
        this.name_index = name_index;
        this.descriptor_index = descriptor_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_NameAndType_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 12)
            throw new ClassFormatError();
        name_index = in.readChar();
        descriptor_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
    }

}
```

A24. **CONSTANT_String_info.java**
Convience class for representing CONSTANT_String_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** String subtype of a constant pool entry.
 */
public final class CONSTANT_String_info extends cp_info{

    /** The index to the actual value of the string. */
```

```
    public int string_index;

    public CONSTANT_String_info(int value) {
        tag = 8;
        string_index = value;
    }

    /** ONLY TO BE USED BY CLASSFILE! */
    public CONSTANT_String_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 8)
            throw new ClassFormatError();
        string_index = in.readChar();
    }

    /** Output serialization, ONLY TO BE USED BY CLASSFILE! */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(string_index);
    }

}
```

A25. **CONSTANT_Utf8_info.java**

Convience class for representing CONSTANT_Utf8_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** Utf8 subtype of a constant pool entry.
 *  We internally represent the Utf8 info byte array
 *  as a String.
 */
public final class CONSTANT_Utf8_info extends cp_info{

    /** Length of the byte array. */
    public int length;

    /** The actual bytes, represented by a String. */
    public String bytes;

    /** This constructor should be used for the purpose
     *  of part creation. It does not set the parent
     *  ClassFile reference.
     */
    public CONSTANT_Utf8_info(String s) {
        tag = 1;
        length = s.length();
        bytes = s;
    }

    /** Used during input serialization by ClassFile only. */
    public CONSTANT_Utf8_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 1)
            throw new ClassFormatError();
        length = in.readChar();
        byte[] b = new byte[length];
```

```
        in.read(b, 0, length);

        // WARNING: String constructor is deprecated.
        bytes = new String(b, 0, length);
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(length);

        // WARNING: Handling of String coversion here might be problematic.
        out.writeBytes(bytes);
    }

    public String toString(){
        return bytes;
    }

}
```

A26. **ConstantValue_attribute.java**

Convience class for representing ConstantValue_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Attribute that allows for initialization of static variables in
 *  classes. This attribute will only reside in a field_info struct.
 */

public final class ConstantValue_attribute extends attribute_info{

    public int constantvalue_index;

    public ConstantValue_attribute(ClassFile cf, int ani, int al, int cvi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        constantvalue_index = cvi;
    }

    public ConstantValue_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        constantvalue_index = in.readChar();
    }

    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(constantvalue_index);
    }

}
```

A27. **cp_info.java**

Convience class for representing cp_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Represents the common interface of all constant pool parts
 *  that all specific constant pool items must inherit from.
 *
 */
public abstract class cp_info{

    /** The type tag that signifies what kind of constant pool
     *  item it is */
    public int tag;

    /** Used for serialization of the object back into a bytestream. */
    abstract void serialize(DataOutputStream out) throws IOException;

    /** Default constructor. Simply does nothing. */
    public cp_info() {}

    /** Constructor simply takes in the ClassFile as a reference to
     *  it's parent
     */
    public cp_info(ClassFile cf) {}

    /** Used during input serialization by ClassFile only. */
    cp_info(ClassFile cf, DataInputStream in)
        throws IOException{
        tag = in.readUnsignedByte();
    }

}
```

A28. **Deprecated_attribute.java**

Convience class for representing Deprecated_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A fix attributed that can be located either in the ClassFile,
 *  field_info or the method_info attribute. Mark deprecated to
 *  indicate that the method, class or field has been superceded.
 */
public final class Deprecated_attribute extends attribute_info{

    public Deprecated_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }


    /** Used during input serialization by ClassFile only. */
    Deprecated_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

A29. **Exceptions_attribute.java**

Convience class for representing Exceptions_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This is the struct where the exceptions table are located.
 *  <br><br>
 *  This attribute can only appear once in a method_info struct.
 */
public final class Exceptions_attribute extends attribute_info{

    public int number_of_exceptions;
    public int[] exception_index_table;

    public Exceptions_attribute(ClassFile cf, int ani, int al, int noe,
                                int[] eit){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_exceptions = noe;
        exception_index_table = eit;
    }

    /** Used during input serialization by ClassFile only. */
    Exceptions_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_exceptions = in.readChar();
        exception_index_table = new int[number_of_exceptions];
        for (int i=0; i<number_of_exceptions; i++)
            exception_index_table[i] = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_exceptions*2);
        super.serialize(out);
        out.writeChar(number_of_exceptions);
        for (int i=0; i<number_of_exceptions; i++)
            out.writeChar(exception_index_table[i]);
    }

}
```

A30. **field_info.java**

Convience class for representing field_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/**   Represents the field_info structure as specified in the JVM
specification.
 */
public final class field_info{

    public int access_flags;
    public int name_index;
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Convenience constructor. */
    public field_info(ClassFile cf, int flags, int ni, int di){
        access_flags = flags;
        name_index = ni;
```

```
        descriptor_index = di;
        attributes_count = 0;
        attributes = new attribute_info[0];
    }

    /** Constructor called only during the serialization process.
     *  <br><br>
     *  This is intentionally left as package protected as we
     *  should not normally call this constructor directly.
     *  <br><br>
     *  Warning: the handling of len is not correct (after String s =...)
     */
    field_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("ConstantValue"))
                attributes[i] = new ConstantValue_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** To serialize the contents into the output format.
     */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A31. InnerClasses_attribute.java

Convience class for representing InnerClasses_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A variable length structure that contains information about an
 *  inner class of this class.
 */
public final class InnerClasses_attribute extends attribute_info{

    public int number_of_classes;
    public classes[] classes;

    public final static class classes{
```

```
        int inner_class_info_index;
        int outer_class_info_index;
        int inner_name_index;
        int inner_class_access_flags;
    }

    public InnerClasses_attribute(ClassFile cf, int ani, int al,
                                  int noc, classes[] c){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_classes = noc;
        classes = c;
    }

    /** Used during input serialization by ClassFile only. */
    InnerClasses_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_classes = in.readChar();
        classes = new InnerClasses_attribute.classes[number_of_classes];
        for (int i=0; i<number_of_classes; i++){
            classes[i] = new classes();
            classes[i].inner_class_info_index = in.readChar();
            classes[i].outer_class_info_index = in.readChar();
            classes[i].inner_name_index = in.readChar();
            classes[i].inner_class_access_flags = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_classes * 8);
        super.serialize(out);
        out.writeChar(number_of_classes);
        for (int i=0; i<number_of_classes; i++){
            out.writeChar(classes[i].inner_class_info_index);
            out.writeChar(classes[i].outer_class_info_index);
            out.writeChar(classes[i].inner_name_index);
            out.writeChar(classes[i].inner_class_access_flags);
        }
    }

}
```

A32. **LineNumberTable_attribute.java**

Convience class for representing LineNumberTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Determines which line of the binary code relates to the
 *  corresponding source code.
 */
public final class LineNumberTable_attribute extends attribute_info{

    public int line_number_table_length;
    public line_number_table[] line_number_table;

    public final static class line_number_table{
        int start_pc;
```

```
        int line_number;
    }

    public LineNumberTable_attribute(ClassFile cf, int ani, int al, int
lntl,
                                        line_number_table[] lnt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        line_number_table_length = lntl;
        line_number_table = lnt;
    }

    /** Used during input serialization by ClassFile only. */
    LineNumberTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        line_number_table_length = in.readChar();
        line_number_table = new
LineNumberTable_attribute.line_number_table[line_number_table_length];
        for (int i=0; i<line_number_table_length; i++){
            line_number_table[i] = new line_number_table();
            line_number_table[i].start_pc = in.readChar();
            line_number_table[i].line_number = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (line_number_table_length * 4);
        super.serialize(out);
        out.writeChar(line_number_table_length);
        for (int i=0; i<line_number_table_length; i++){
            out.writeChar(line_number_table[i].start_pc);
            out.writeChar(line_number_table[i].line_number);
        }
    }

}
```

A33. **LocalVariableTable_attribute.java**

Convience class for representing LocalVariableTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Used by debugger to find out how the source file line number is linked
 *  to the binary code. It has many to one correspondence and is found in
 *  the Code_attribute.
 */
public final class LocalVariableTable_attribute extends attribute_info{

    public int local_variable_table_length;
    public local_variable_table[] local_variable_table;

    public final static class local_variable_table{
        int start_pc;
        int length;
        int name_index;
        int descriptor_index;
        int index;
```

```
    }

    public LocalVariableTable_attribute(ClassFile cf, int ani, int al,
                                         int lvtl, local_variable_table[]
lvt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        local_variable_table_length = lvtl;
        local_variable_table = lvt;
    }

    /** Used during input serialization by ClassFile only. */
    LocalVariableTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        local_variable_table_length = in.readChar();
        local_variable_table = new
LocalVariableTable_attribute.local_variable_table[local_variable_table_lengt
h];
        for (int i=0; i<local_variable_table_length; i++){
            local_variable_table[i] = new local_variable_table();
            local_variable_table[i].start_pc = in.readChar();
            local_variable_table[i].length = in.readChar();
            local_variable_table[i].name_index = in.readChar();
            local_variable_table[i].descriptor_index = in.readChar();
            local_variable_table[i].index = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (local_variable_table_length * 10);
        super.serialize(out);
        out.writeChar(local_variable_table_length);
        for (int i=0; i<local_variable_table_length; i++){
            out.writeChar(local_variable_table[i].start_pc);
            out.writeChar(local_variable_table[i].length);
            out.writeChar(local_variable_table[i].name_index);
            out.writeChar(local_variable_table[i].descriptor_index);
            out.writeChar(local_variable_table[i].index);
        }
    }

}
```

A34. **method_info.java**

Convience class for representing method_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This follows the method_info in the JVM specification.
 */
public final class method_info {

    public int access_flags;
    public int name_index;
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;
```

```
    /** Constructor. Creates a method_info, initializes it with
     *  the flags set, and the name and descriptor indexes given.
     *  A new uninitialized code attribute is also created, and stored
     *  in the <i>code</i> variable.*/
    public method_info(ClassFile cf, int flags, int ni, int di,
                       int ac, attribute_info[] a) {
        access_flags = flags;
        name_index = ni;
        descriptor_index = di;
        attributes_count = ac;
        attributes = a;
    }

    /** This method creates a method_info from the current pointer in the
     *  data stream. Only called by during the serialization of a complete
     *  ClassFile from a bytestream, not normally invoked directly.
     */
    method_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("Code"))
                attributes[i] = new Code_attribute(cf, in);
            else if (s.equals("Exceptions"))
                attributes[i] = new Exceptions_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** Output serialization of the method_info to a byte array.
     *  Not normally invoked directly.
    */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A35. **SourceFile_attribute.java**

Convience class for representing SourceFile_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A SourceFile attribute is an optional fixed_length attribute in
 *  the attributes table. Only located in the ClassFile struct only
 *  once.
```

```
 */
public final class SourceFile_attribute extends attribute_info{

    public int sourcefile_index;

    public SourceFile_attribute(ClassFile cf, int ani, int al, int sfi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        sourcefile_index = sfi;
    }

    /** Used during input serialization by ClassFile only. */
    SourceFile_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        sourcefile_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(sourcefile_index);
    }

}
```

A36. **Synthetic_attribute.java**

Convience class for representing Synthetic_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A synthetic attribute indicates that this class does not have
 *  a generated code source. It is likely to imply that the code
 *  is generated by machine means rather than coded directly. This
 *  attribute can appear in the classfile, method_info or field_info.
 *  It is fixed length.
 */
public final class Synthetic_attribute extends attribute_info{

    public Synthetic_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }

    /** Used during output serialization by ClassFile only. */
    Synthetic_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

ANNEXURE B

B1

Method <clinit>

0 new #2 <Class test>
3 dup
4 invokespecial #3 <Method test()>
7 putstatic #4 <Field test thisTest>
10 return

B2

Method <clinit>
0 invokestatic #3 <Method boolean isAlreadyLoaded()>
3 ifeq 7
6 return
7 new #5 <Class test>
10 dup
11 invokespecial #6 <Method test()>
14 putstatic #7 <Field test thisTest>
17 return

B3

Method <init>
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 aload_0
5 invokestatic #2 <Method long currentTimeMillis()>
8 putfield #3 <Field long timestamp>
11 return

B4

Method <init>
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 invokestatic #2 <Method boolean isAlreadyLoaded()>
7 ifeq 13
10 return
11 aload_0
12 invokestatic #4 <Method long currentTimeMillis()>
15 putfield #5 <Field long timestamp>
18 return

B5

Method <clinit>
0 ldc #2 <String "test">
2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
5 ifeq 9
8 return
9 new #5 <Class test>
12 dup
13 invokespecial #6 <Method test()>
16 putstatic #7 <Field test thisTest>
19 return

B6

Method <init>
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>

70~

```
4 aload_0
5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
8 ifeq 13
11 return
12 aload_0
13 invokestatic #4 <Method long currentTimeMillis()>
16 putfield #5 <Field long timestamp>
19 return
```

ANNEXURE B7

This excerpt is the source-code of InitClient, which queries an "initialisation server" for the initialisation status of the relevant class or object.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitClient{


   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT = 20;


   /** InitServer network values. */
   public final static String serverAddress =
      System.getProperty("InitServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("InitServer_network_port"));


   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();


   /** Called when a object is being initialized. */
   public static boolean isAlreadyLoaded(Object o){

      // First of all, we need to resolve the globalID
      // for object 'o'.  To do this we use the hashCodeToGlobalID
      // table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{
```

```
// Next, we want to connect to the InitServer, which will inform us
// of the initialization status of this object.
Socket socket = new Socket(serverAddress, serverPort);
DataOutputStream out =
   new DataOutputStream(socket.getOutputStream());
DataInputStream in =
   new DataInputStream(socket.getInputStream());

// Ok, now send the serialized request to the InitServer.
out.writeInt(INITIALIZE_OBJECT);
out.writeInt(globalID);
out.flush();

// Now wait for the reply.
int status = in.readInt();      // This is a blocking call.  So we
                                // will wait until the remote side
                                // sends something.

if (status == NACK){
   throw new AssertionError(
      "Negative acknowledgement. Request failed.");
}else if (status != ACK){
   throw new AssertionError("Unknown acknowledgement: "
      + status + ". Request failed.");
}

// Next, read in a 32bit argument which is the count of previous
// initializations.
int count = in.readInt();

// If the count is equal to 0, then this is the first
// initialization, and hence isAlreadyLoaded should be false.
// If however, the count is greater than 0, then this is already
// initialized, and thus isAlreadyLoaded should be true.
boolean isAlreadyLoaded = (count == 0 ? false : true);

// Close down the connection.
out.writeInt(CLOSE);
out.flush();
out.close();
in.close();

socket.close();         // Make sure to close the socket.
```

```
            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }


      /** Called when a class is being initialized. */
      public static boolean isAlreadyLoaded(String name){

         try{

            // First of all, we want to connect to the InitServer, which will
            // inform us of the initialization status of this class.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
               new DataOutputStream(socket.getOutputStream());
            DataInputStream in =
               new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the InitServer.
            out.writeInt(INITIALIZE_CLASS);
            out.writeInt(name.length());          // A 32bit length argument of
                                                  // the String name.
            out.write(name.getBytes(), 0, name.length());      // The byte-
                                                               // encoded
                                                               // String name.
            out.flush();

            // Now wait for the reply.
            int status = in.readInt();      // This is a blocking call. So we
                                            // will wait until the remote side
                                            // sends something.

            if (status == NACK){
               throw new AssertionError(
                  "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }
```

```
            // Next, read in a 32bit argument which is the count of the
            // previous intializations.
            int count = in.readInt();

            // If the count is equal to 0, then this is the first
            // initialization, and hence isAlreadyLoaded should be false.
            // If however, the count is greater than 0, then this is already
            // loaded, and thus isAlreadyLoaded should be true.
            boolean isAlreadyLoaded = (count == 0 ? false : true);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();        // Make sure to close the socket.

            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }

}
```

ANNEXURE B8

This excerpt is the source-code of InitServer, which receives an initialisation status query by InitClient and in response returns the corresponding status.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitServer implements Runnable{

   /** Protocol specific values */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT= 20;
```

```
/** InitServer network values. */
public final static int serverPort = 20001;

/** Table of initialization records. */
public final static Hashtable initializations = new Hashtable();

/** Private input/output objects. */
private Socket socket = null;
private DataOutputStream outputStream;
private DataInputStream inputStream;
private String address;

public static void main(String[] s)
throws Exception{

   System.out.println("InitServer_network_address="
      + InetAddress.getLocalHost().getHostAddress());
   System.out.println("InitServer_network_port=" + serverPort);

   // Create a serversocket to accept incoming initialization operation
   // connections.
   ServerSocket serverSocket = new ServerSocket(serverPort);

   while (!Thread.interrupted()){

      // Block until an incoming initialization operation connection.
      Socket socket = serverSocket.accept();

      // Create a new instance of InitServer to manage this
      // initialization operation connection.
      new Thread(new InitServer(socket)).start();

   }

}

/** Constructor. Initialize this new InitServer instance with necessary
    resources for operation. */
public InitServer(Socket s){
   socket = s;
   try{
      outputStream = new DataOutputStream(s.getOutputStream());
      inputStream = new DataInputStream(s.getInputStream());
```

```
            address = s.getInetAddress().getHostAddress();
        }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
        }
    }
    /** Main code body. Decode incoming initialization operation requests and
        execute accordingly. */
    public void run(){

        try{

            // All commands are implemented as 32bit integers.
            // Legal commands are listed in the "protocol specific values"
            // fields above.
            int command = inputStream.readInt();

            // Continue processing commands until a CLOSE operation.
            while (command != CLOSE){

                if (command == INITIALIZE_CLASS){        // This is an
                                                         // INITIALIZE_CLASS
                                                         // operation.

                    // Read in a 32bit length field 'l', and a String name for
                    // this class of length 'l'.
                    int length = inputStream.readInt();
                    byte[] b = new byte[length];
                    inputStream.read(b, 0, b.length);
                    String className = new String(b, 0, length);

                    // Synchronize on the initializations table in order to
                    // ensure thread-safety.
                    synchronized (initializations){

                        // Locate the previous initializations entry for this
                        // class, if any.
                        Integer entry = (Integer) initializations.get(className);

                        if (entry == null){    // This is an unknown class so
                                               // update the table with a
                                               // corresponding entry.

                            initializations.put(className, new Integer(1));
```

```
            // Send a positive acknowledgement to InitClient,
            // together with the count of previous initializations
            // of this class - which in this case of an unknown
            // class must be 0.
            outputStream.writeInt(ACK);
            outputStream.writeInt(0);
            outputStream.flush();

         }else{               // This is a known class, so update
                              // the count of initializations.

            initializations.put(className,
               new Integer(entry.intValue() + 1));

            // Send a positive acknowledgement to InitClient,
            // together with the count of previous initializtions
            // of this class - which in this case of a known class
            // must be the value of "entry.intValue()".
            outputStream.writeInt(ACK);
            outputStream.writeInt(entry.intValue());
            outputStream.flush();

         }

      }

   }else if (command == INITIALIZE_OBJECT){  // This is an
                                             // INITIALIZE_OBJECT
                                             // operation.

      // Read in the globalID of the object to be initialized.
      int globalID = inputStream.readInt();

      // Synchronize on the initializations table in order to
      // ensure thread-safety.
      synchronized (initializations){

         // Locate the previous initializations entry for this
         // object, if any.
         Integer entry = (Integer) initializations.get(
            new Integer(globalID));

         if (entry == null){    // This is an unknown object so
                                // update the table with a
```

```
                                        // corresponding entry.

                  initializations.put(new Integer(globalID),
                     new Integer(1));

                  // Send a positive acknowledgement to InitClient,
                  // together with the count of previous initializations
                  // of this object - which in this case of an unknown
                  // object must be 0.
                  outputStream.writeInt(ACK);
                  outputStream.writeInt(0);
                  outputStream.flush();

               }else{         // This is a known object so update the
                              // count of initializations.

                  initializations.put(new Integer(globalID),
                     new Integer(entry.intValue() + 1));

                  // Send a positive acknowledgement to InitClient,
                  // together with the count of previous initializations
                  // of this object - which in this case of a known
                  // object must be value "entry.intValue()".
                  outputStream.writeInt(ACK);
                  outputStream.writeInt(entry.intValue());
                  outputStream.flush();

               }

            }

         }else{         // Unknown command.
            throw new AssertionError(
               "Unknown command. Operation failed.");
         }

         // Read in the next command.
         command = inputStream.readInt();

      }

   }catch (Exception e){
      throw new AssertionError("Exception: " + e.toString());
   }finally{
```

```
         try{
            // Closing down. Cleanup this connection.
            outputStream.flush();
            outputStream.close();
            inputStream.close();
            socket.close();
         }catch (Throwable t){
            t.printStackTrace();
         }
         // Garbage these references.
         outputStream = null;
         inputStream = null;
         socket = null;
      }
   }


}
```

ANNEXURE B9

This excerpt is the source-code of the example application used in the before/after examples of Annexure B.

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public static example currentExample;

   /** Shared instance field. */
   public long timestamp;

   /** Static intializer. (clinit) */
   static{

      currentExample = new example();

   }

   /** Instance intializer (init) */
   public example(){

      timestamp = System.currentTimeMillis();
```

}

}

ANNEXURE B10

InitLoader.java

This excerpt is the source-code of InitLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class InitLoader extends URLClassLoader{


   public InitLoader(URL[] urls){
      super(urls);
   }


   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{
         BufferedInputStream in = new
            BufferedInputStream(findResource(name.replace('.',
            '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}


      for (int i=0; i<cf.methods_count; i++){

         // Find the <clinit> method_info struct.
         String methodName = cf.constant_pool[
            cf.methods[i].name_index].toString();
         if (!methodName.equals("<clinit>")){
            continue;
         }

         // Now find the Code_attribute for the <clinit> method.
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute)
cf.methods[i].attributes[j];

            // First, shift the code[] down by 4 instructions.
            byte[][] code2 = new byte[ca.code.length+4][];
            System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
            ca.code = code2;

            // Then enlarge the constant_pool by 7 items.
            cp_info[] cpi = new cp_info[cf.constant_pool.length+7];
            System.arraycopy(cf.constant_pool, 0, cpi, 0,
               cf.constant_pool.length);
            cf.constant_pool = cpi;
```

```
            cf.constant_pool_count += 7;

            // Now add the constant pool items for these instructions,
starting
            // with String.
            CONSTANT_String_info csi = new CONSTANT_String_info(

      ((CONSTANT_Class_info)cf.constant_pool[cf.this_class]).name_index);
            cf.constant_pool[cf.constant_pool.length-7] = csi;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("InitClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
               new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("isAlreadyLoaded");
            cf.constant_pool[cf.constant_pool.length-4] = u1;

            // Next add the second UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("(Ljava/lang/String;)Z");
            cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
            CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
               cf.constant_pool.length-4, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and
NameAndType.
            CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
               cf.constant_pool.length-5, cf.constant_pool.length-2);
            cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code,
starting
            // with LDC.
            ca.code[0] = new byte[3];
            ca.code[0][0] = (byte) 19;
            ca.code[0][1] = (byte) (((cf.constant_pool.length-7) >> 8) &
0xff);
            ca.code[0][2] = (byte) ((cf.constant_pool.length-7) & 0xff);

            // Now Add the INVOKESTATIC instruction.
            ca.code[1] = new byte[3];
            ca.code[1][0] = (byte) 184;
            ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) &
0xff);
            ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

            // Next add the IFEQ instruction.
            ca.code[2] = new byte[3];
            ca.code[2][0] = (byte) 153;
            ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
            ca.code[2][2] = (byte) (4 & 0xff);

            // Finally, add the RETURN instruction.
            ca.code[3] = new byte[1];
            ca.code[3][0] = (byte) 177;

            // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH
values.
```

```
            ca.code_length += 10;
            ca.attribute_length += 10;


         }
      }


      try{

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         cf.serialize(out);

         byte[] b = out.toByteArray();

         return defineClass(name, b, 0, b.length);

      }catch (Exception e){
       e.printStackTrace();
         throw new ClassNotFoundException(name);
      }

   }

}
```

CLAIMS

1. A multiple computer system having at least one application program running simultaneously on a plurality of computers interconnected by a communications network, wherein a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

2. The system as claimed in claim 1 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers creates an object in that computer then the created object is propagated by the distributed run time means of said one computer to all the other computers.

3. The system as claimed in claim 2 wherein each said application program is modified before, during, or after loading by inserting an initialization routine to modify each instance at which said application program creates an object, said initialization routine propagating every object newly created by one computer to all said other computers.

4. The system as claimed in claim 3 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

5. The system as claimed in claim 2 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

6. A plurality of computers interconnected via a communications link and operating at least one application program simultaneously wherein each said computer in operating said at least one application program creates objects

only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

7. The plurality of computers as claimed in claim 6 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

8. The plurality of computers as claimed in claim 6 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

9. The plurality of computers as claimed in claim 6 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

10. A method of running at least one application program on a plurality of computers simultaneously, said computers being interconnected by means of a communications network, said method comprising the steps of:
(i) creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and
(ii) creating the initial contents of each of said identically named objects substantially the same.

11. The method as claimed in claim 10 comprising the further step of,
(iii) if a portion of said application program running on one of said computers creates an object in that computer, then the created object is propagated to all of the other computers via said communications network.

12. The method as claimed in claim 11 including the further step of:
(iv) modifying said application program before, during or after loading by inserting an initialization routine to modify each instance at which said

application program creates an object, said initialization routine propagating every object created by one computer to all said other computers.

13. The method as claimed in claim 12 including the further step of:

(v) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

14. The method as claimed in claim 12 including the further step of:

(vi) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

15. A method of operating at least one application program simultaneously on a plurality of computers all interconnected via a communications link and each having at least a minimum predetermined local memory capacity, said method comprising the steps of:

(i) initially providing each local memory in substantially identical condition,

(ii) satisfying all requests to create objects generated by said application program in said local memory, and

(iii) communicating via said communications link all said objects created at each said computer and which reside locally to all the remainder of said plurality of computers whereby the objects of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

16. The method as claimed in claim 15 including the further step of:

(iv) communicating said locally created objects constituting an updating data transmission at a data transfer rate which is substantially less than the local memory read rate.

17. A method of compiling or modifying an application program to run simultaneously on a plurality of computers interconnected via a communications link, said method comprising the steps of:
(i) detecting instructions which create objects utilizing one of said computers,
(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

18. The method as claimed in claim 17 and carried out prior to loading the application program onto each said computer, or during loading of the application program onto each said computer, or after loading of the application program onto each said computer and before execution of the relevant portion of the application program.

19. In a multiple thread processing computer operation in which individual threads of a single application program are simultaneously being processed each on a corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

20. The improvement as claimed in claim 19 wherein objects created in the memory associated with one said thread are communicated by the computer of said one thread to all other said computers.

21. The improvement as claimed in claim 19 wherein objects created the memory associated with one said thread are transmitted to the computer associated with another said thread and are transmitted thereby to all said other computers.

22. A method of ensuring consistent initialization of an application program to be run simultaneously on a plurality of computers interconnected via a communications network, said method comprising the steps of:
(i) scrutinizing said application program at, or prior to, or after loading to

detect each program step defining an initialization routine, and

(ii) modifying said initialization routine to ensure consistent operation of all said computers.

23. The method claimed in claim 22 wherein step (ii) comprises the steps of:

(iii) loading and executing said initialization routine on one of said computers,

(iv) modifying said initialization routine by said one computer, and

(v) transferring said modified initialization routine to each of the remaining computers.

24. The method as claimed in claim 23 wherein said modified initialization routine is supplied by said one computer direct to each of said remaining computers.

25. The method as claimed in claim 23 wherein said modified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

26. The method claimed in claim 22 wherein step (ii) comprises the steps of:

(vi) loading and modifying said initialization routine on one of said computers,

(vii)said one computer sending said unmodified initialization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said initialization routine after receipt of same.

27. The method claimed in claim 26 wherein said unmodified initialization routine is supplied by said one computer directly to each of said remaining computers.

28. The method claimed in claim 26 wherein said unmodified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

29. A computer program product comprising a set of program instructions stored in a storage medium and operable to permit a plurality of computers to carry out the method as claimed in claim 10 or 15 or 17 or 22.

30. A plurality of computers interconnected via a communication network and operable to ensure consistent initialization of an application program running simultaneously of said computers, said computers being programmed to carry out the method as claimed in claim 10 or 15 or 17 or 22 or being loaded with the computer program product as claimed in claim 29.

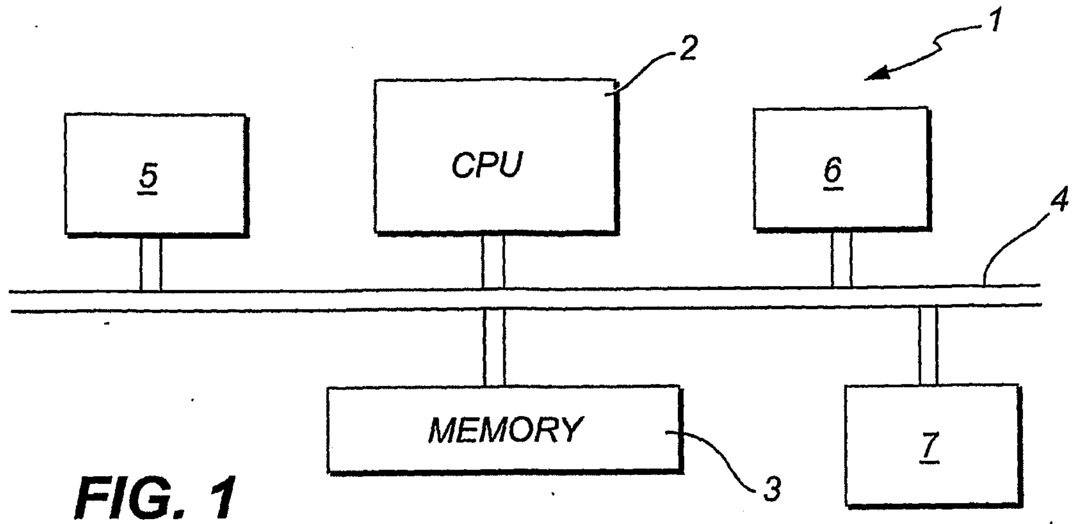


FIG. 1
PRIOR ART

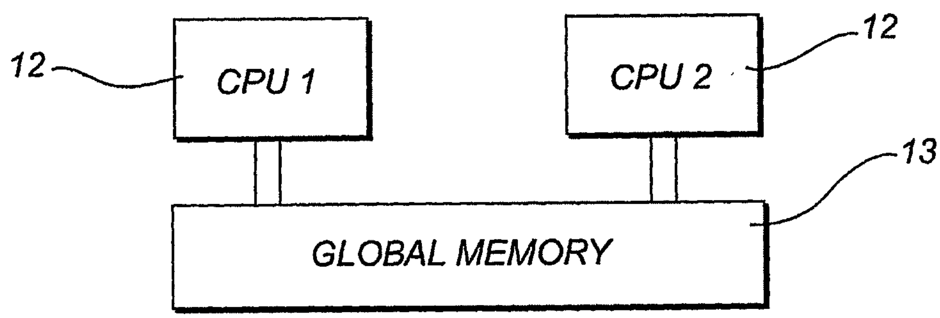


FIG. 2
PRIOR ART

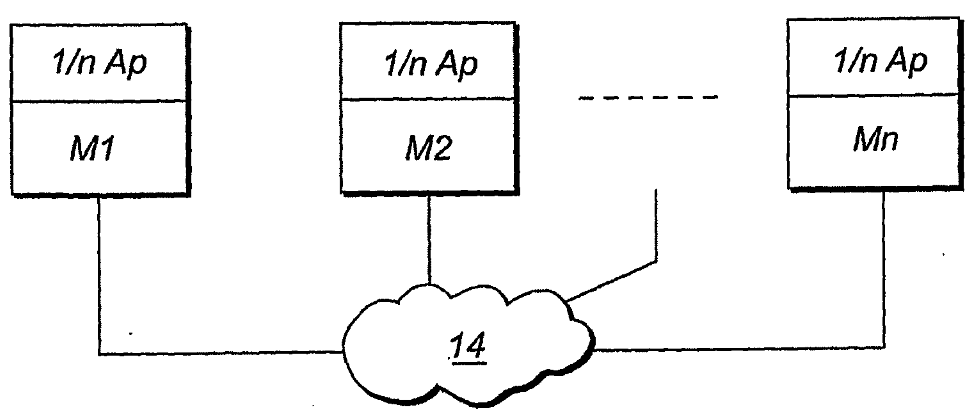


FIG. 3
PRIOR ART

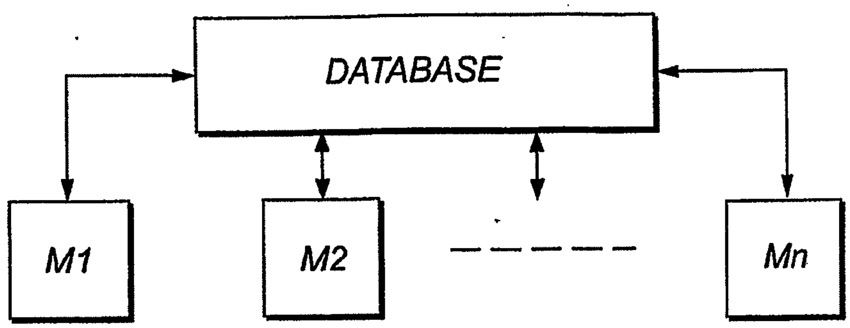


FIG. 4
PRIOR ART

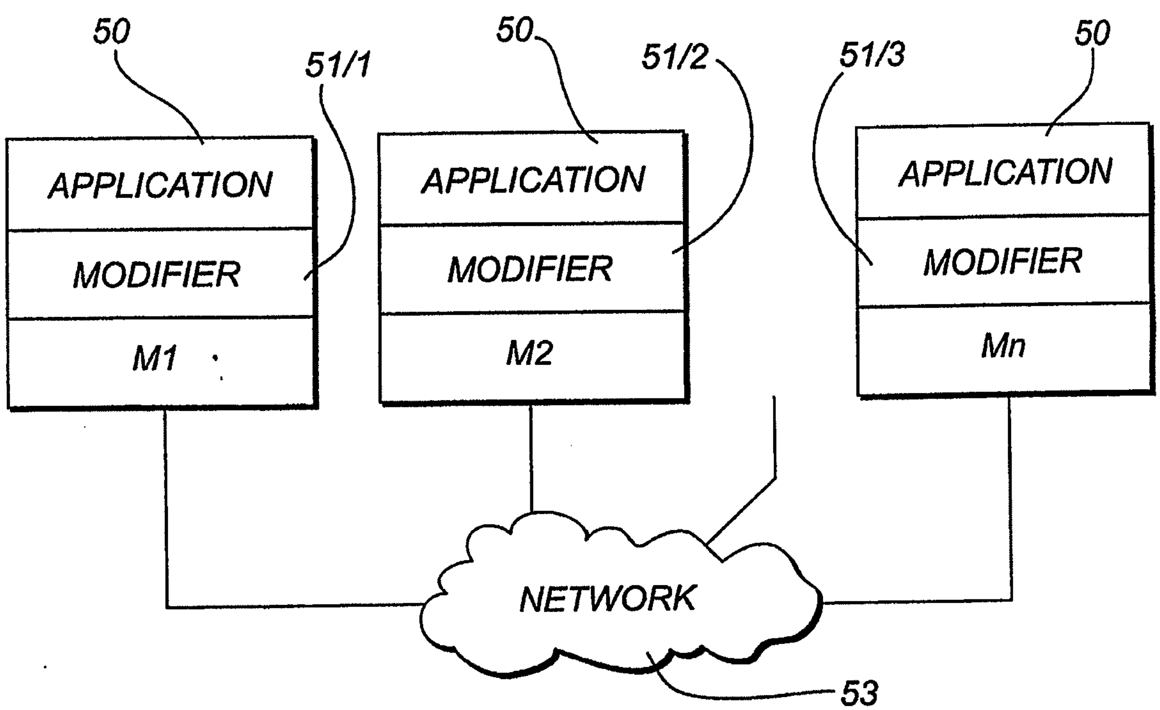


FIG. 5

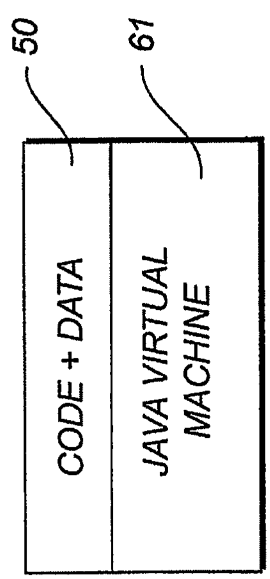


FIG. 6
PRIOR ART

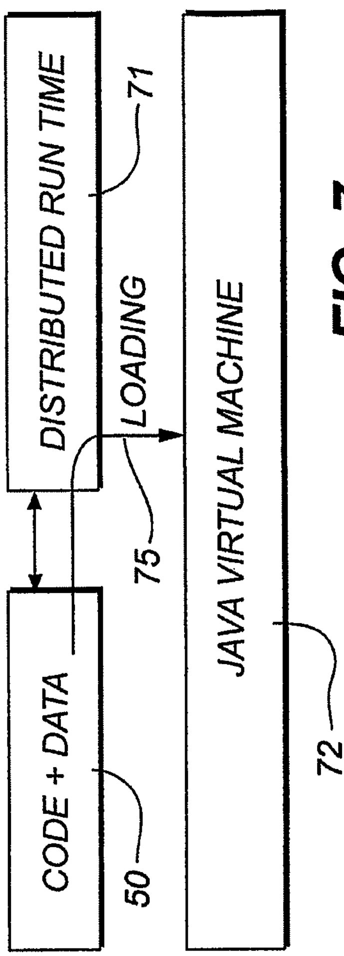


FIG. 7

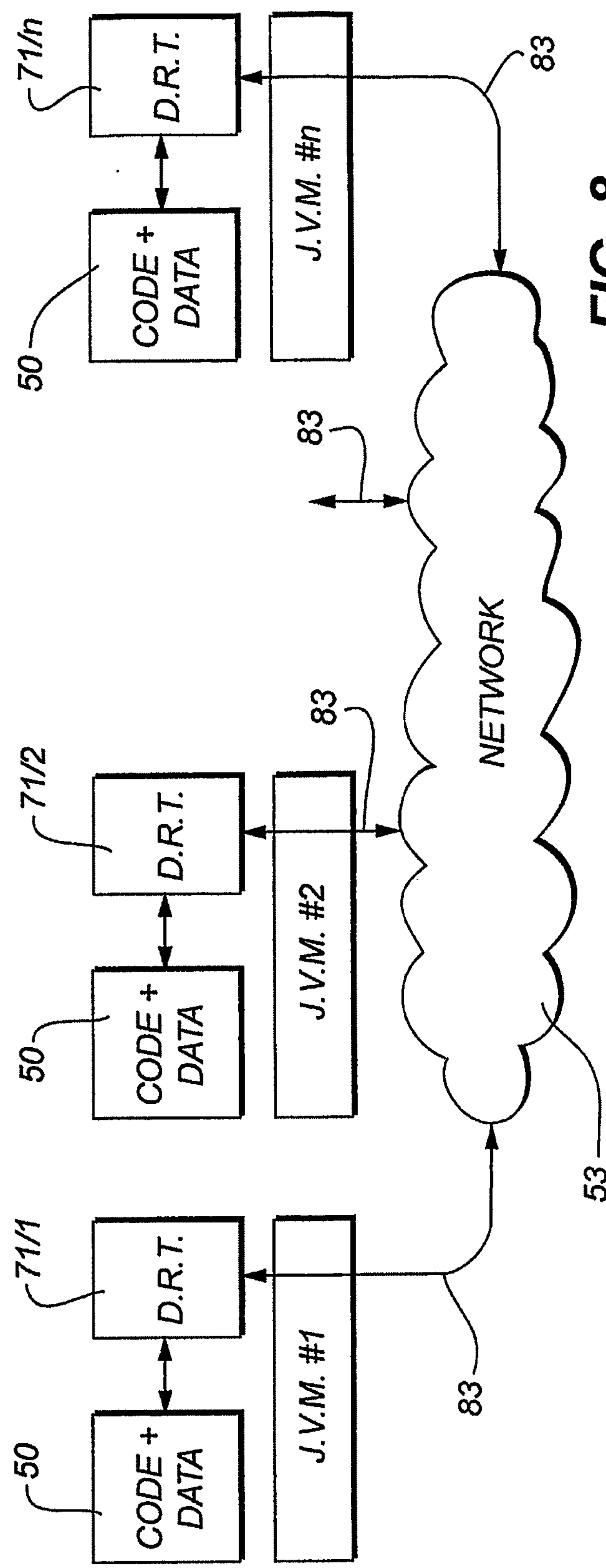


FIG. 8

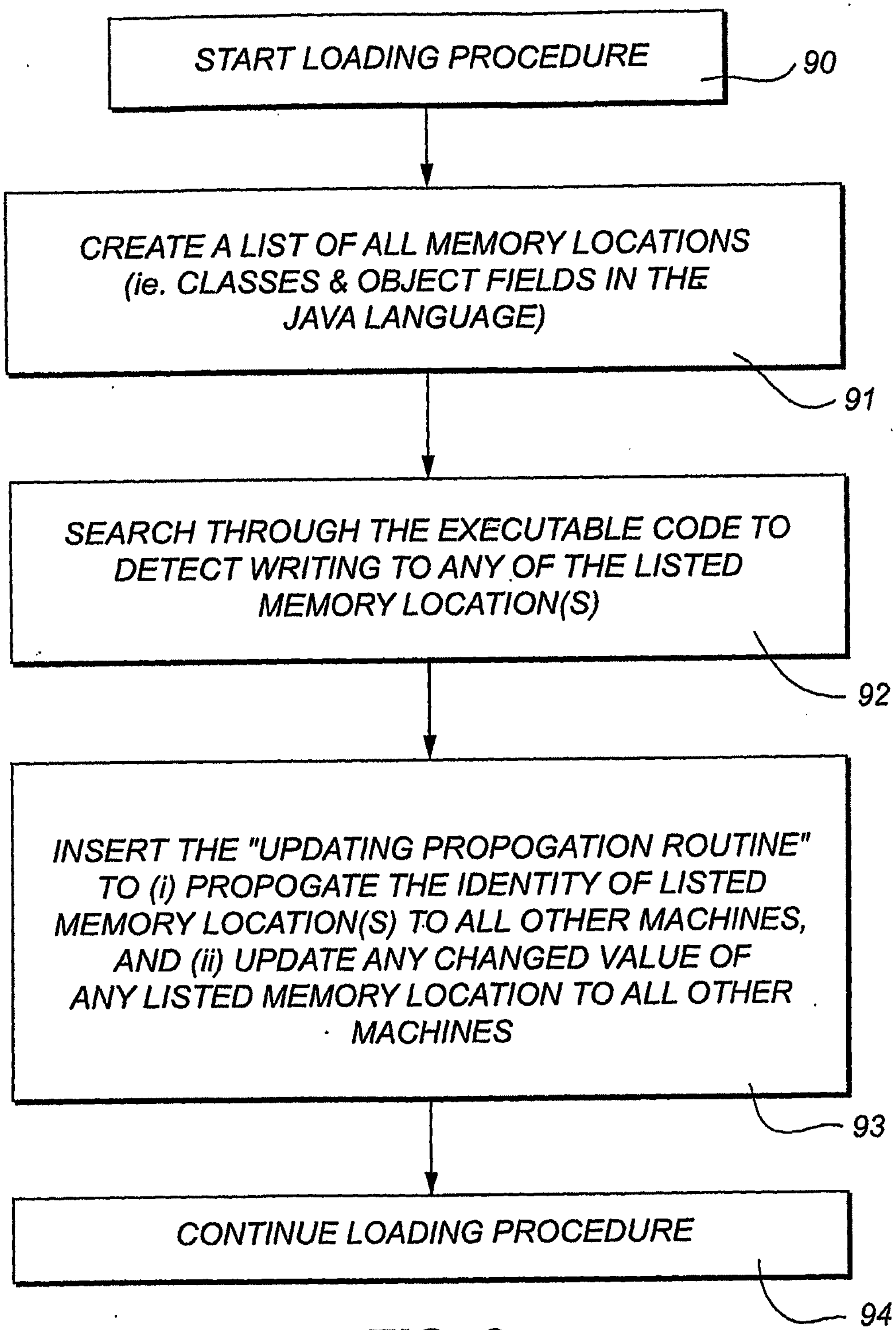

START LOADING PROCEDURE
90
CREATE A LIST OF ALL MEMORY LOCATIONS (ie. CLASSES & OBJECT FIELDS IN THE JAVA LANGUAGE)
91
SEARCH THROUGH THE EXECUTABLE CODE TO DETECT WRITING TO ANY OF THE LISTED MEMORY LOCATION(S)
92
INSERT THE "UPDATING PROPOGATION ROUTINE" TO (i) PROPOGATE THE IDENTITY OF LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND (ii) UPDATE ANY CHANGED VALUE OF ANY LISTED MEMORY LOCATION TO ALL OTHER MACHINES
93
CONTINUE LOADING PROCEDURE
94

FIG. 9

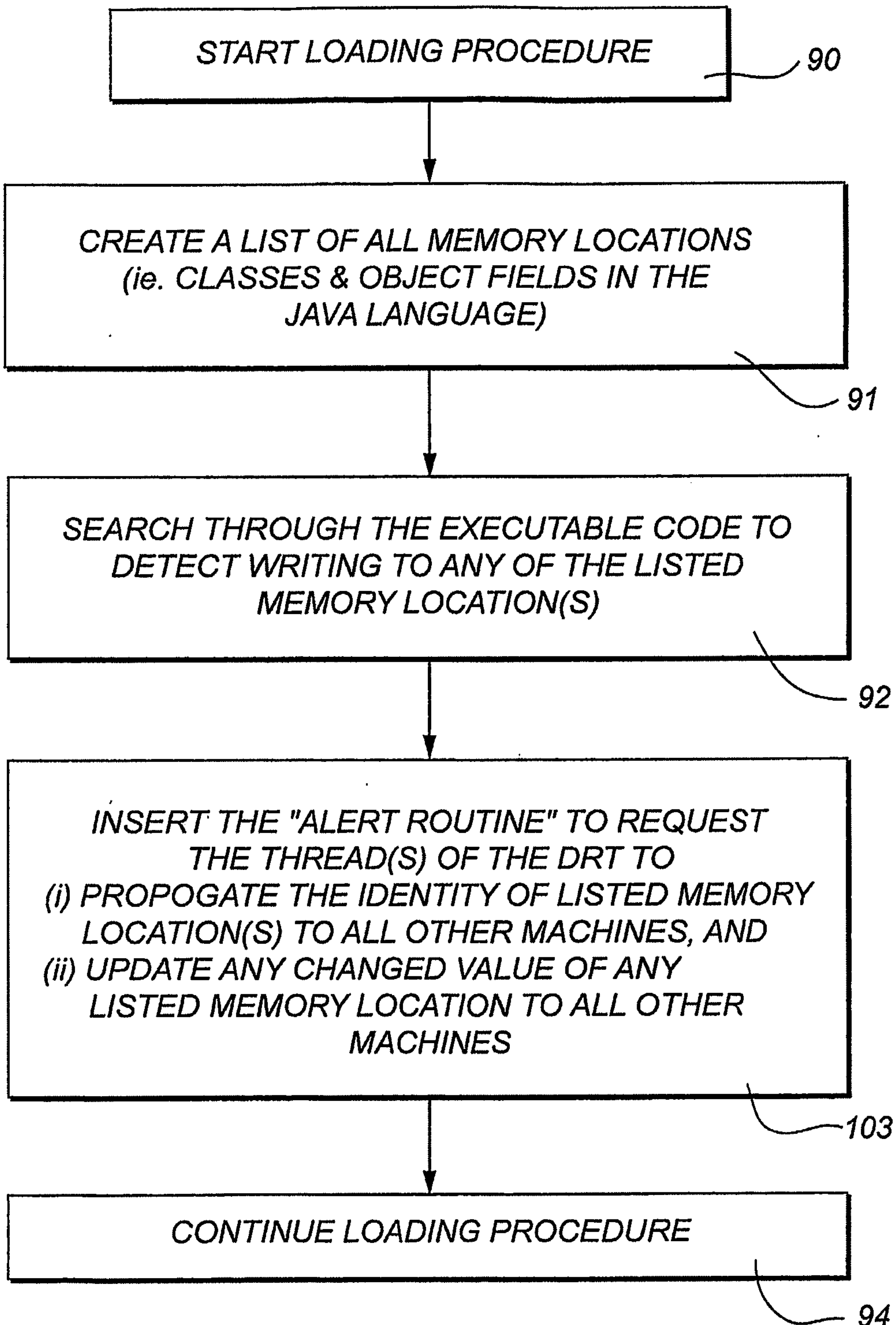
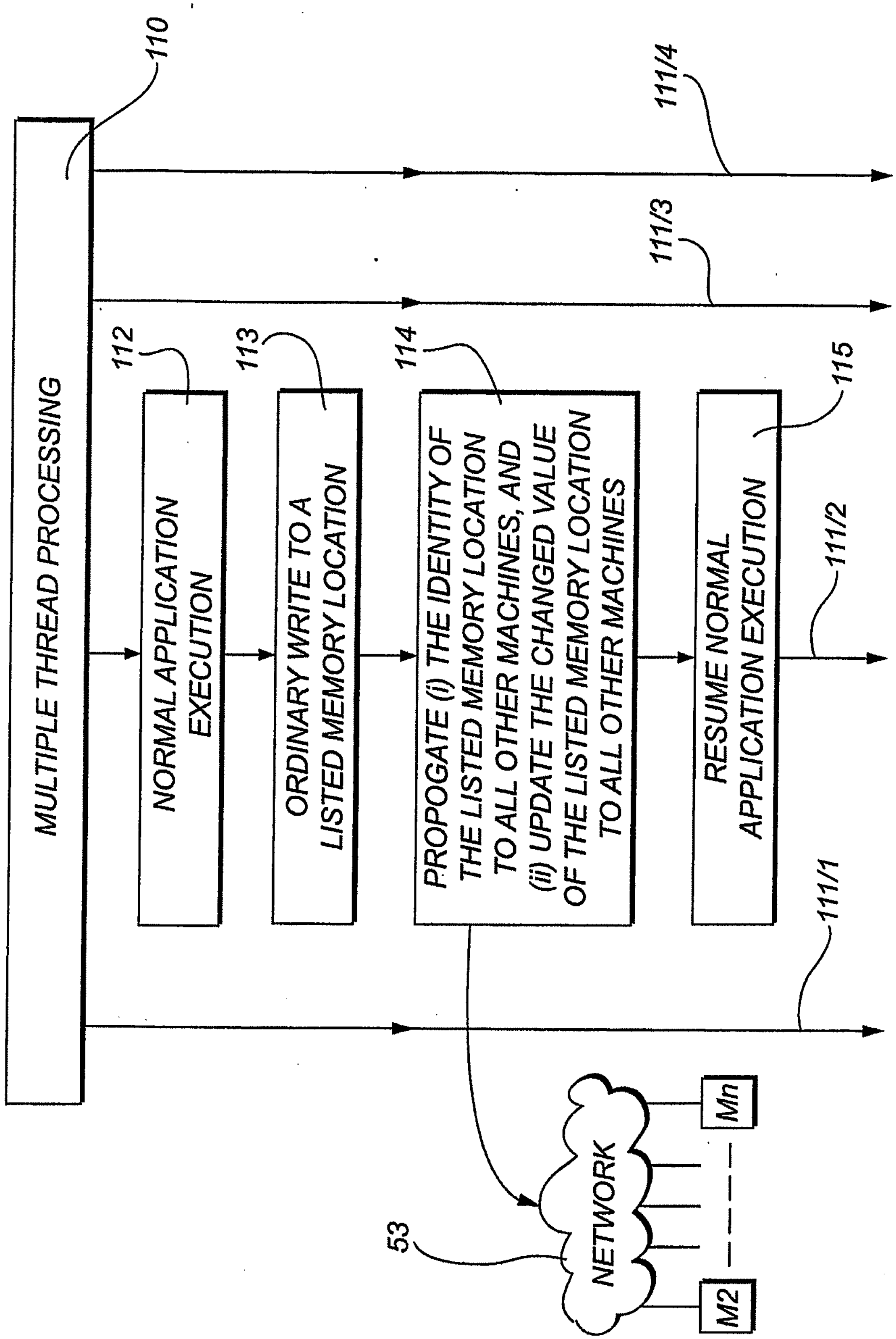

START LOADING PROCEDURE
90
CREATE A LIST OF ALL MEMORY LOCATIONS (ie. CLASSES & OBJECT FIELDS IN THE JAVA LANGUAGE)
91
SEARCH THROUGH THE EXECUTABLE CODE TO DETECT WRITING TO ANY OF THE LISTED MEMORY LOCATION(S)
92
INSERT THE "ALERT ROUTINE" TO REQUEST THE THREAD(S) OF THE DRT TO (i) PROPOGATE THE IDENTITY OF LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND (ii) UPDATE ANY CHANGED VALUE OF ANY LISTED MEMORY LOCATION TO ALL OTHER MACHINES
103
CONTINUE LOADING PROCEDURE
94

FIG. 10



110
MULTIPLE THREAD PROCESSING
112
NORMAL APPLICATION EXECUTION
113
ORDINARY WRITE TO A LISTED MEMORY LOCATION
114
PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
115
RESUME NORMAL APPLICATION EXECUTION
111/1
111/2
111/3
111/4
53
NETWORK
M2
Mn

FIG.11

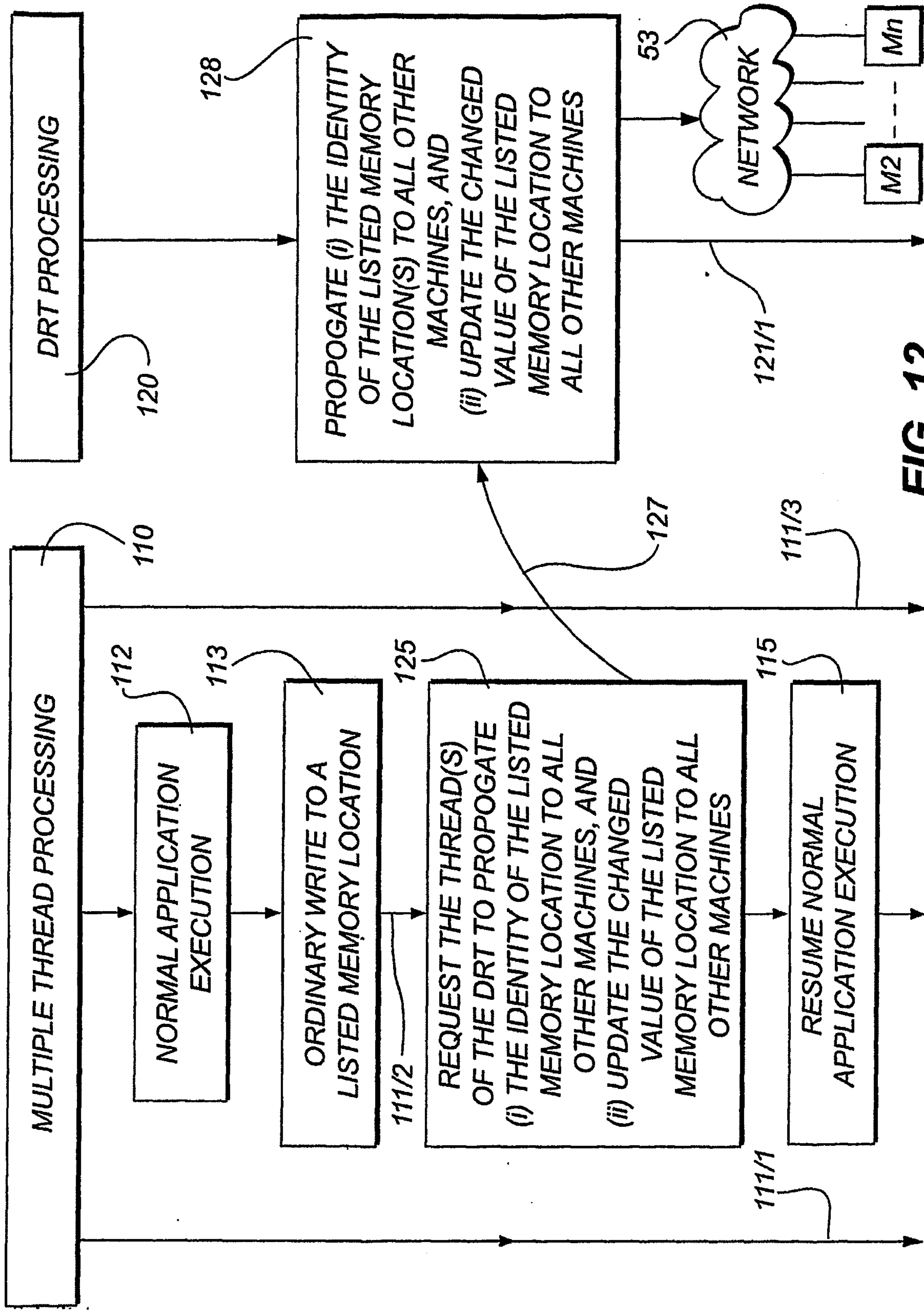

MULTIPLE THREAD PROCESSING
110
112
NORMAL APPLICATION EXECUTION
113
ORDINARY WRITE TO A LISTED MEMORY LOCATION
111/2
125
REQUEST THE THREAD(S) OF THE DRT TO PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
115
RESUME NORMAL APPLICATION EXECUTION
111/1
111/3
127
DRT PROCESSING
120
128
PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
121/1
53
NETWORK
M2
Mn

FIG. 12

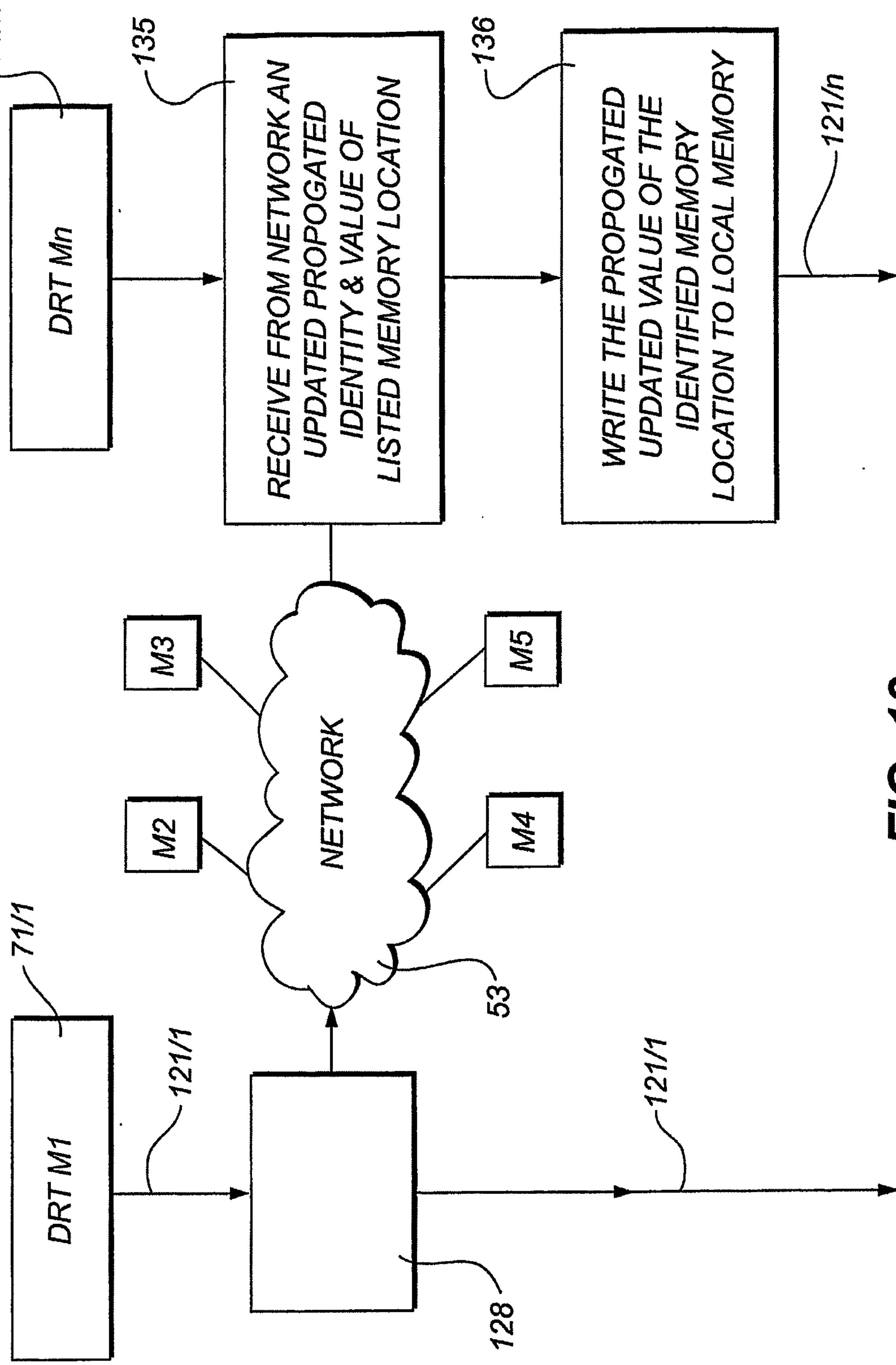

71/n
DRT Mn
135
RECEIVE FROM NETWORK AN UPDATED PROPOGATED IDENTITY & VALUE OF LISTED MEMORY LOCATION
136
WRITE THE PROPOGATED UPDATED VALUE OF THE IDENTIFIED MEMORY LOCATION TO LOCAL MEMORY
121/n
M3
M5
NETWORK
M2
M4
53
71/1
DRT M1
121/1
121/1
128

FIG. 13

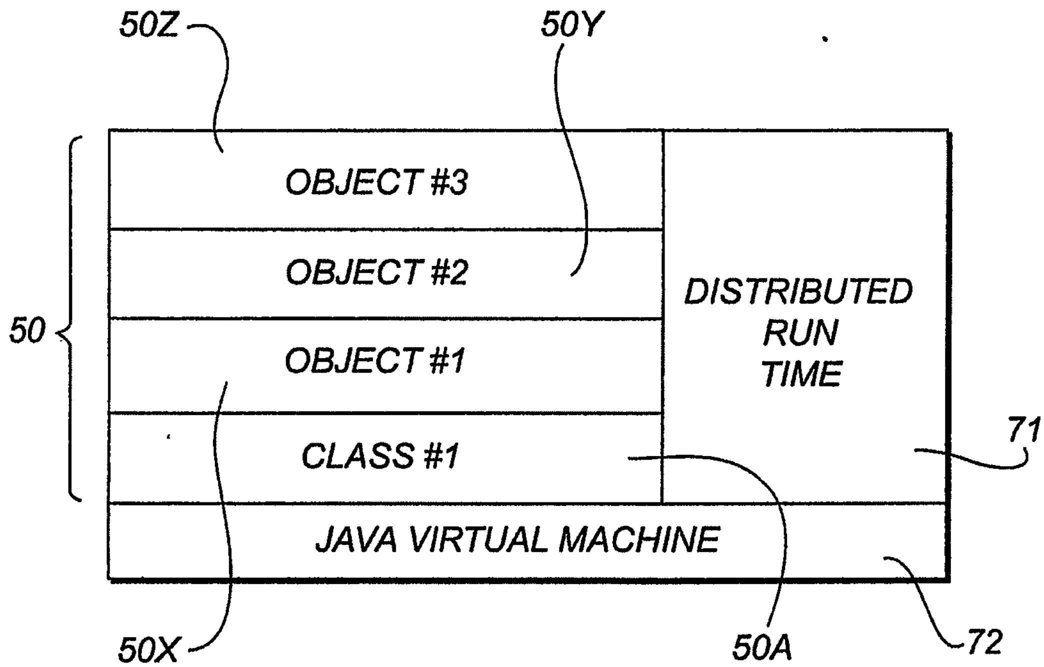


FIG. 14
PRIOR ART

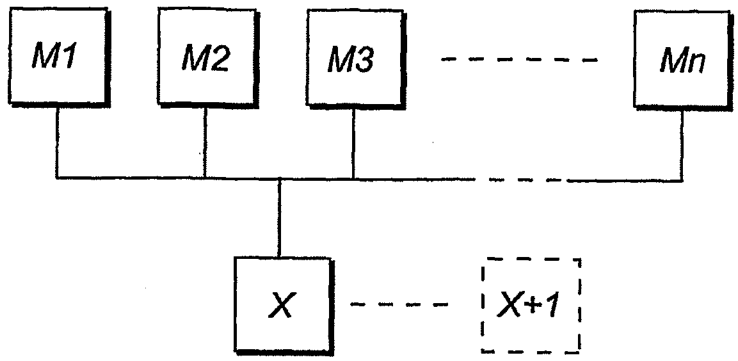

M1
M2
M3
Mn
X
X+1

FIG. 15

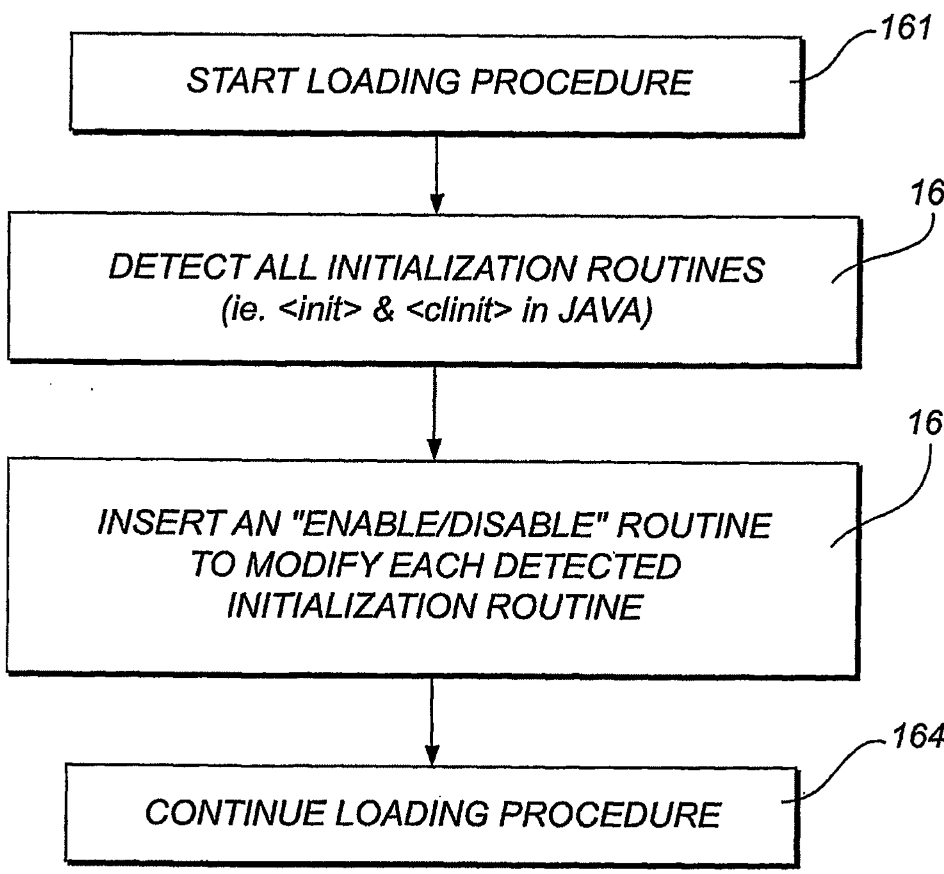

START LOADING PROCEDURE
161
DETECT ALL INITIALIZATION ROUTINES
(ie. <init> & <clinit> in JAVA)
162
INSERT AN "ENABLE/DISABLE" ROUTINE
TO MODIFY EACH DETECTED
INITIALIZATION ROUTINE
163
CONTINUE LOADING PROCEDURE
164

FIG. 16

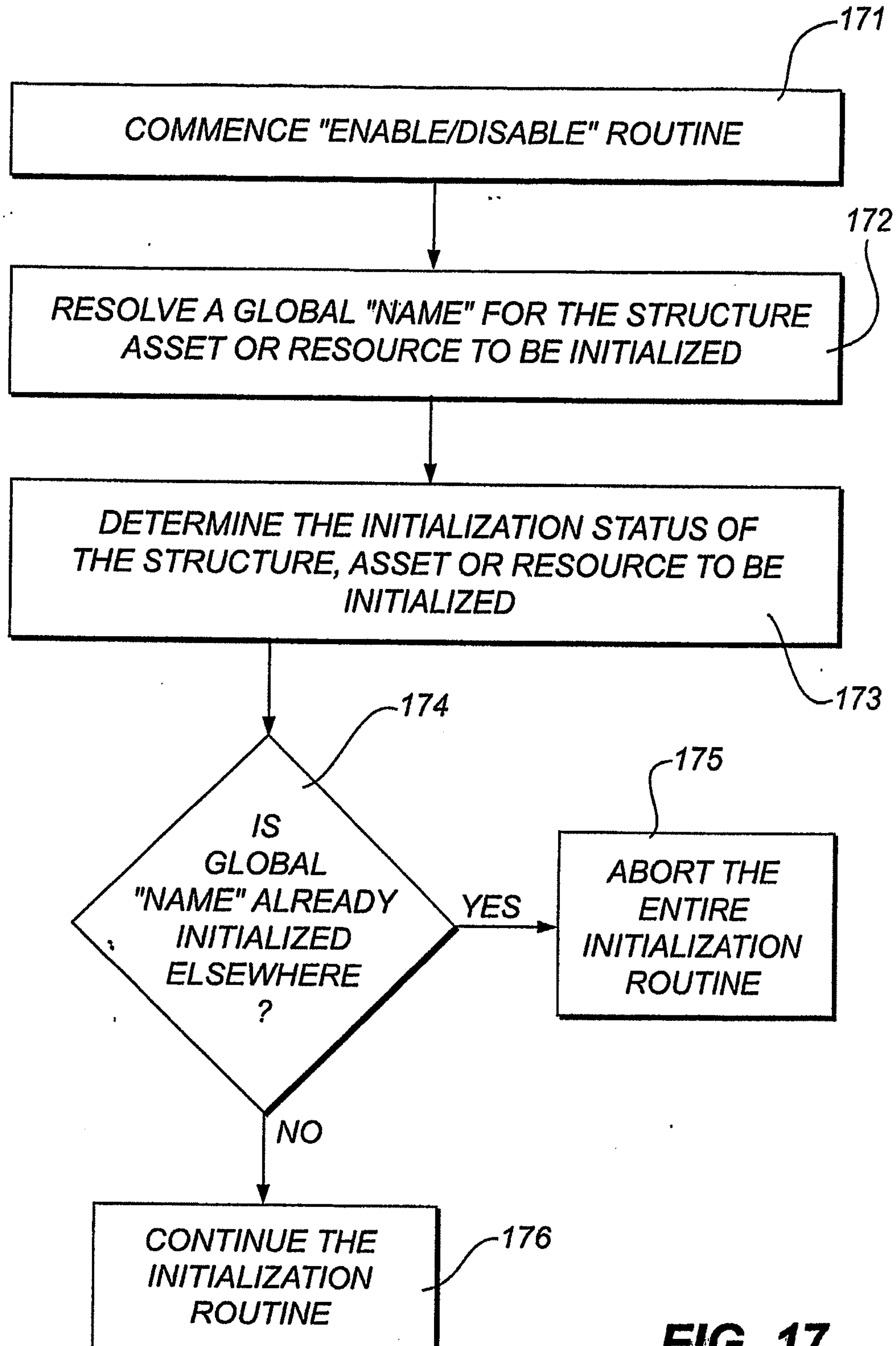

171
COMMENCE "ENABLE/DISABLE" ROUTINE
172
RESOLVE A GLOBAL "NAME" FOR THE STRUCTURE ASSET OR RESOURCE TO BE INITIALIZED
DETERMINE THE INITIALIZATION STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE INITIALIZED
173
174
IS GLOBAL "NAME" ALREADY INITIALIZED ELSEWHERE ?
YES
175
ABORT THE ENTIRE INITIALIZATION ROUTINE
NO
CONTINUE THE INITIALIZATION ROUTINE
176

FIG. 17

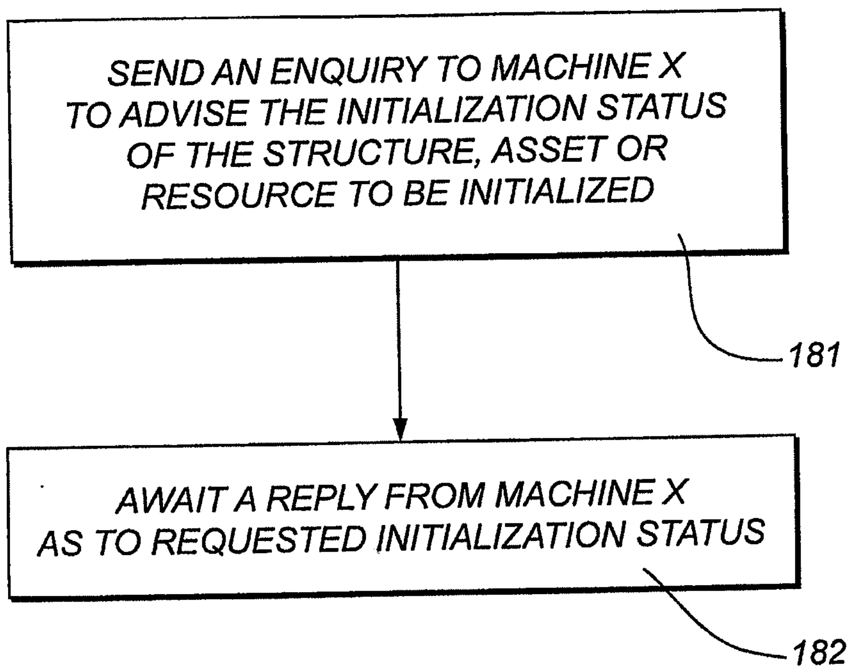

SEND AN ENQUIRY TO MACHINE X TO ADVISE THE INITIALIZATION STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE INITIALIZED
181
AWAIT A REPLY FROM MACHINE X AS TO REQUESTED INITIALIZATION STATUS
182

FIG. 18

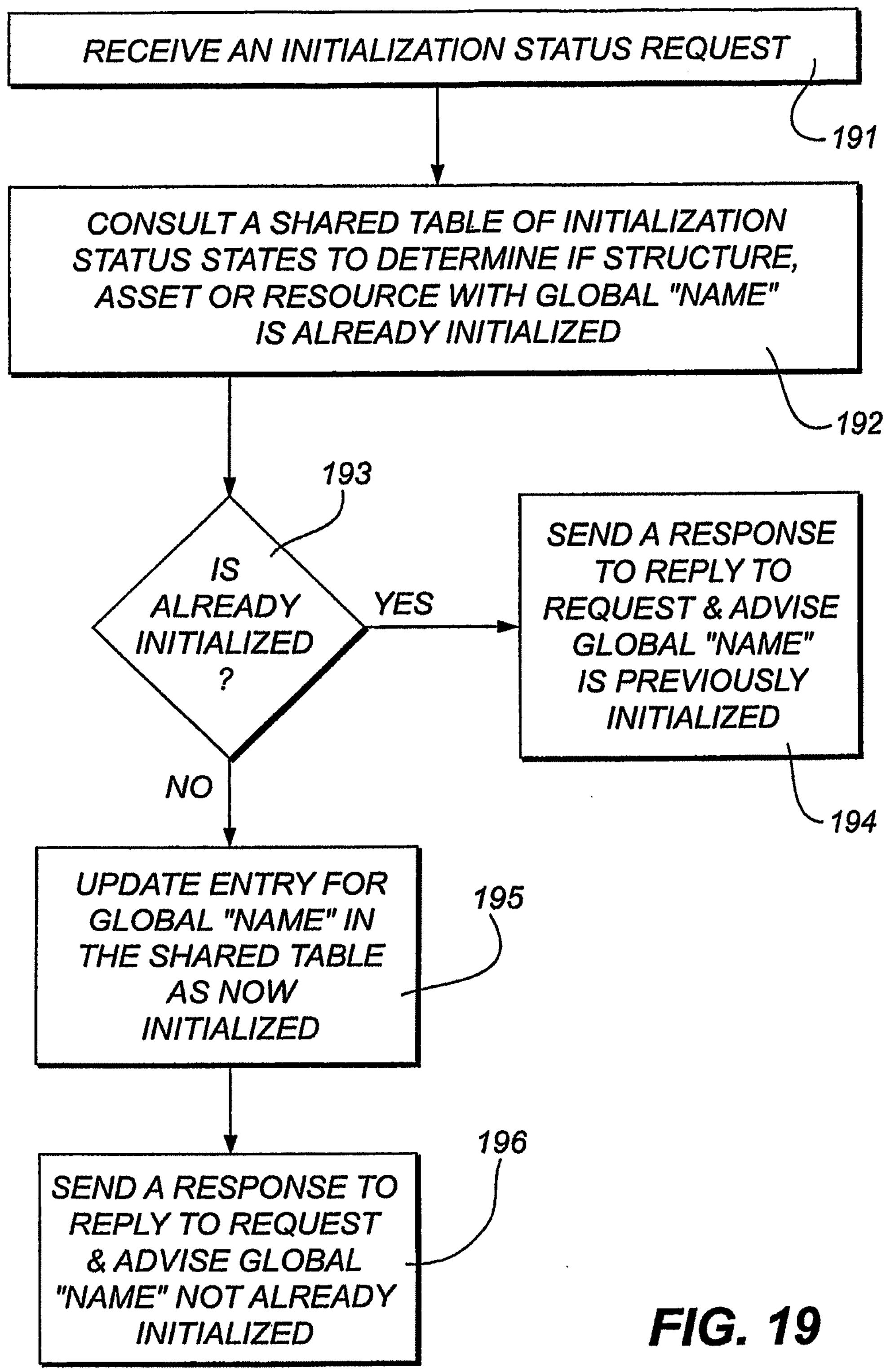

RECEIVE AN INITIALIZATION STATUS REQUEST
191
CONSULT A SHARED TABLE OF INITIALIZATION STATUS STATES TO DETERMINE IF STRUCTURE, ASSET OR RESOURCE WITH GLOBAL "NAME" IS ALREADY INITIALIZED
192
193
IS ALREADY INITIALIZED ?
YES
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" IS PREVIOUSLY INITIALIZED
194
NO
UPDATE ENTRY FOR GLOBAL "NAME" IN THE SHARED TABLE AS NOW INITIALIZED
195
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" NOT ALREADY INITIALIZED
196

FIG. 19

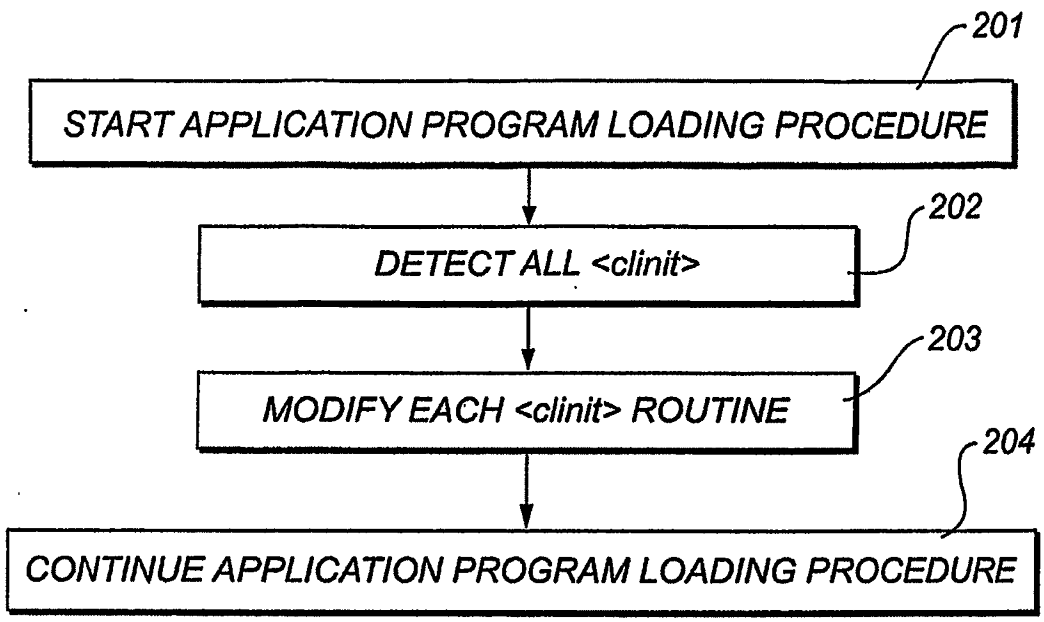

201
START APPLICATION PROGRAM LOADING PROCEDURE
202
DETECT ALL <clinit>
203
MODIFY EACH <clinit> ROUTINE
204
CONTINUE APPLICATION PROGRAM LOADING PROCEDURE

FIG. 20

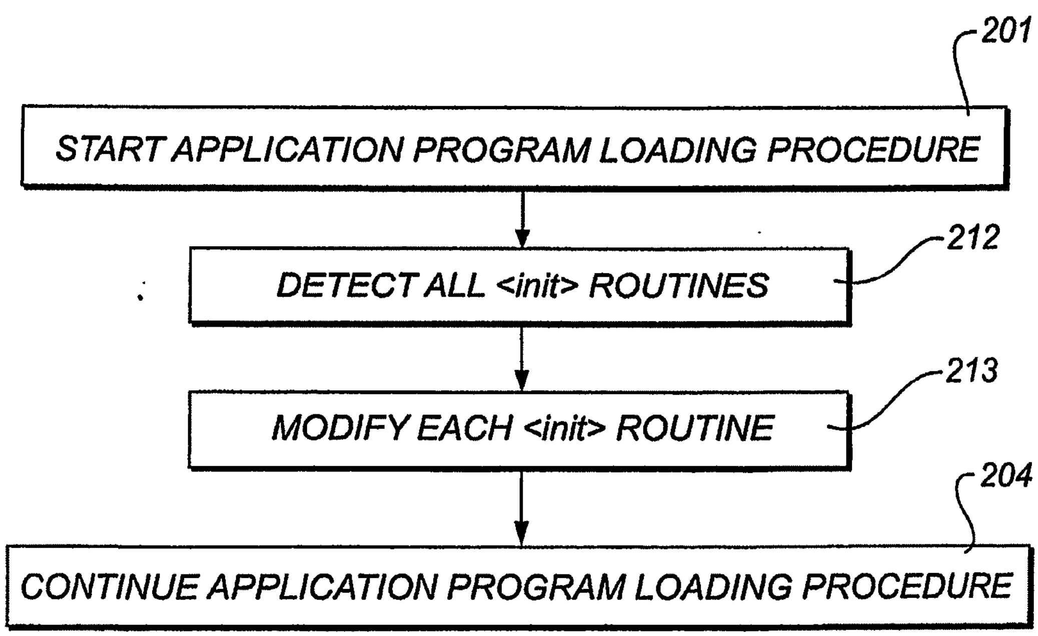

201
START APPLICATION PROGRAM LOADING PROCEDURE
212
DETECT ALL <init> ROUTINES
213
MODIFY EACH <init> ROUTINE
204
CONTINUE APPLICATION PROGRAM LOADING PROCEDURE

FIG. 21

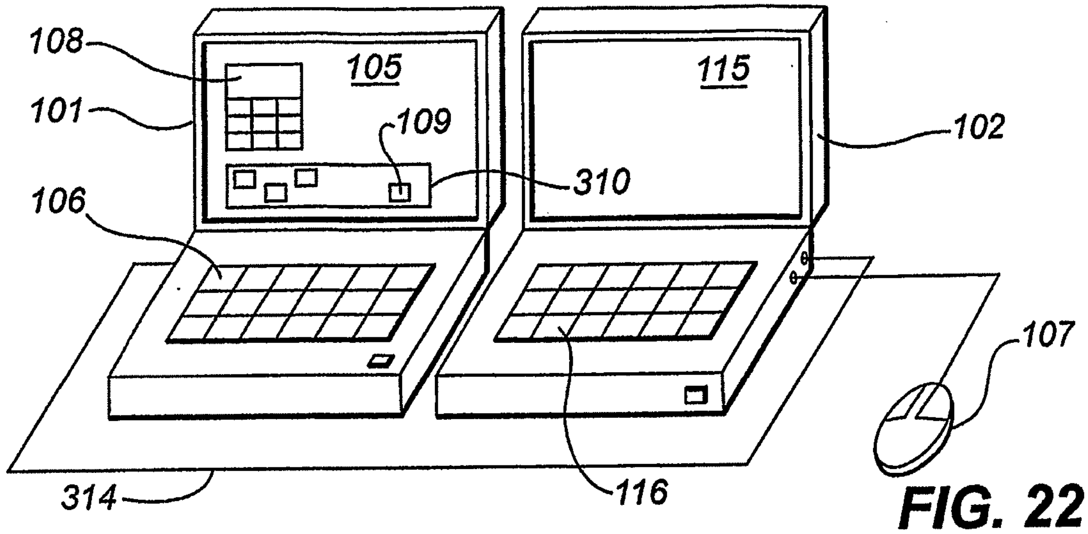


FIG. 22

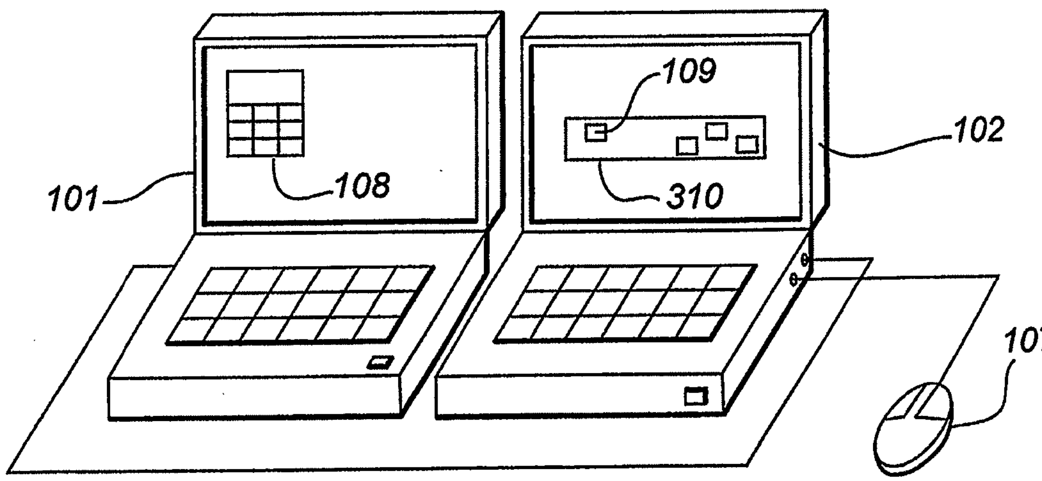


FIG. 23

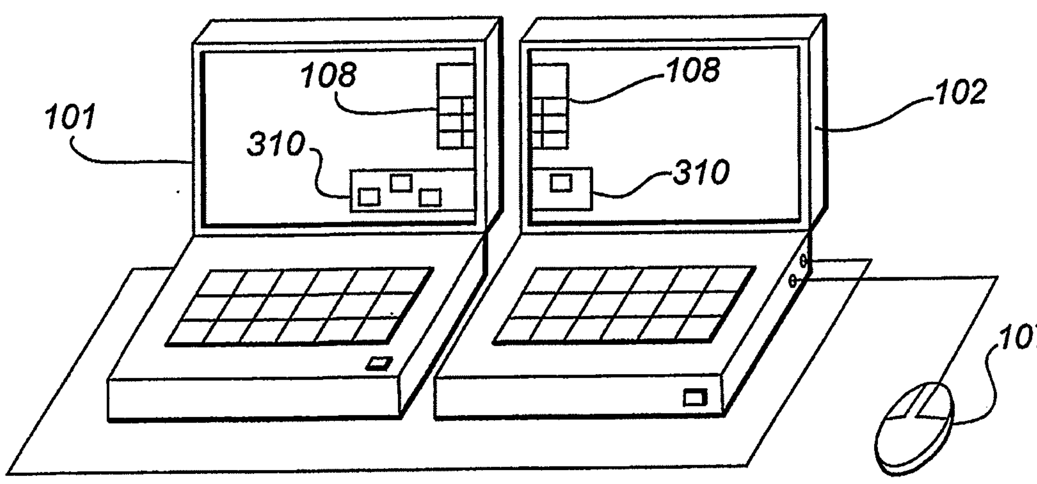


FIG. 24

INTERNATIONAL SEARCH REPORT	International application No.
	PCT/AU2005/000578

A. CLASSIFICATION OF SUBJECT MATTER

Int. Cl. [7]: G06F 15/16

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
DWPI, USPTO, PCT, IEEE, internet (object, replicate, redundant, mirror, cluster, distributed, lock, thread, compile, high availability, runtime, virtual machine, etc.)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 2003/083614 A1 (ETERNAL SYSTEMS, INC.), 9 October 2003 the whole document	1-30
X	US 6,625,751 B1 (STAROVIC et al), 23 September 2003 the whole document	1-30
X	WO 2002/044835 A2 (GINGERICH), 6 June 2002 the whole document	1-30
X	US 2004/0073828 A1 (BRONSTEIN), 15 April 2004 the whole document	1-30

[X] Further documents are listed in the continuation of Box C [X] See patent family annex

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
4 July 2005	11 JUL 2005
Name and mailing address of the ISA/AU	Authorized officer
AUSTRALIAN PATENT OFFICE PO BOX 200, WODEN ACT 2606, AUSTRALIA E-mail address: pct@ipaustralia.gov.au Facsimile No. (02) 6285 3929	**MATTHEW HOLLINGWORTH** Telephone No : (02) 6283 2024

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,488,723 A (BARADEL et al), 30 January 1996 the whole document	1-30
X	T. C. Bressoud, *TFT: A Software System for Application-Transparent Fault Tolerance* Proc. 28th Annual International Symposium on Fault-Tolerant Computing, pp. 128-37, 1998	1-30

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report		Patent Family Member					
WO	03083614	AU	2003223352	AU	2003230748	EP	1495414
		EP	1495571	US	2004078617	US	2004078618
		WO	03084116				
US	6625751	GB	2353113				
WO	0244835	AU	32411/02				
US	2004073828	NONE					
US	5488723	EP	0572307	FR	2691559	WO	9324884

Due to data integration issues this family listing may not include 10 digit Australian applications filed since May 2001.

END OF ANNEX